AEROSPACE REPORT NO.
ATR-92(2778)-6

SDVS Verification of a Stage 3 Ada Program

30 September 1992

Prepared by

J. E. DONER
Computer Systems Division

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Engineering and Technology Group

THE AEROSPACE CORPORATION
El Segundo, California

Aerospace Report No.
ATR-92(2778)-6

SDVS VERIFICATION OF A STAGE 3 ADA PROGRAM

Prepared by

J. E. Doner
Computer Systems Division

Accesion For		
NTIS CRA&I		☒
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

30 September 1992

Engineering and Technology Group
THE AEROSPACE CORPORATION
El Segundo, CA 90245-4691

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Report No.
ATR-92(2778)-6

SDVS VERIFICATION OF A STAGE 3 ADA PROGRAM

Prepared

John E. Doner

J. E. Doner

Approved

Beth Levy

B. H. Levy, Manager
Computer Assurance Section

DB Baker

D. B. Baker, Director
Trusted Computer Systems Department

DB Baker for CA Sunshine

C. A. Sunshine, Principal Director
Computer Science and Technology
Subdivision

Abstract

We describe an SDVS correctness proof for a fragment of operational code. This code implements a minor variant of the familiar bubble-sort algorithm, and uses `for`-loops and the `record` structure—Ada features that are either new with this version of the SDVS translator, or not previously exercised extensively. The proof demonstrates these features of SDVS, and is interesting because of the techniques used and the light it throws on possible improvements and enhancements for SDVS. We also discuss some data security problems and the ability of SDVS to treat them.

Contents

1 Introduction

As part of continuing efforts to develop the software and hardware verification capabilities of SDVS, we exercise new features by constructing proofs for example programs. This year we sought program fragments from operational systems, as opposed to writing example programs on our own. The subject of this report is one such program fragment and its corresponding verification.

We requested from a source with access to operational code a program fragment for which a verification would be of some interest. The program we received used long, descriptive identifiers that are meaningful in the original context, such as this typical assignment statement:

```
DIMPOINT_COUNTS_BY_SECTOR(SECTOR_INDEX):=DIMPOINT_COUNTS_BY_SECTOR(SECTOR_NUMBER);
```

The choice of identifier names is irrelevant to program structure and verification, so we began our analysis by systematically changing identifiers to more convenient ones. Further, it was necessary to restructure the program fragment into a form convenient for verification; we did this by essentially encapsulating the bulk of the code as a procedure declared internally to a testing procedure, which has declarations to create a suitable environment for the code fragment, including the needed data types and a global variable for the array to be sorted. The result of these changes is the following:

```
        -- with text_io; use text_io;
-- with integer_io; use integer_io;

procedure testctrsort is

        type R_TYPE is
                record
                        a   : integer;
                        b   : integer;
                        c   : integer;
                end record;

        SS: constant integer := 100;

        type ARRAY_TYPE is array (0..SS-1) of R_TYPE;
        X :  ARRAY_TYPE;

        procedure ctrsort(N, S : IN integer) is
                i : integer;
                temp : R_TYPE;
        begin
                -- contractor sort
                for j in 0..N-1 loop
                        i := j;
                        while i < S-1 loop
                                i := i+1;
                                if X(j).b > X(i).b then
                                        temp:=X(j);
                                        X(j):=X(i);
                                        X(i):=temp;
                                end if;
                        end loop;
                end loop;
        end ctrsort;

begin
        null;
end testctrsort;
```

The nested loops which are the body of the procedure named `ctrsort` (for "contractor sort") constitute the part of interest for verification. The outer procedure `testctrsort` exists merely to provide the framework for declaring `ctrsort`.[1] It is a minor variation of the familiar bubble-sort algorithm that appears in most elementary programming textbooks, the only difference being a provision for stopping the process at a point where the array is only partially sorted. Our source cited as questions of interest the following:

- the correctness of the program fragment
- whether the program fragment made access to locations outside the boundary of the array

We are able to prove the correctness of the fragment, and this proof is the main subject of this report. The second question will be addressed in Section 4, where we discuss the ability of SDVS to handle this and related data-security questions.

This program (namely, the procedure `ctrsort`) is a bubble sort of an array of records, each of which has three integer fields, with the second field as the sorting key. The parameter S should be the length of the array to be sorted, and N a natural number $\leq$ S. [2]If N = S, the program sorts the entire array, but if N < S, the program merely puts the least N elements of the array in order at the beginning of the array.

Thus, the `ctrsort` program sorts an array of records using a standard bubble-sort technique, except that it allows for a partial completion of the process in which only an initial subset of the records is sorted. The correctness proof is not dependent in any essential way upon the use of records, so we adopt a further simplification of the program in which the array of records is replaced by an array of integers. For technical reasons involving the internals of SDVS, this has the desirable side-effect of significantly reducing the run time for the correctness proof.[3]Thus, the actual program for which we will give a detailed discussion of the correctness proof is the procedure `pblsort`[4] in the following:

[1]It may seem curious that the body of procedure `testctrsort` is vacuous (consists only of a `null` statement. Note however that we have no need to execute `testctrsort`; we only want to prove correctness of the procedure `ctrsort` declared internally to it. `testctrsort` is merely a vehicle for defining the environment necessary to declare `ctrsort`. The body is present only because it is required for correct Ada syntax.

[2]These conditions are within the original intent of the procedure, and are included in the result we finally prove about the procedure, but are not specifically checked for within the procedure. Of course, this could be done, but there is little to be gained by it so far as our correctness proof is concerned.

[3]The reduction is significant, and the reasons why it occurs are currently under investigation.

[4]The "pbl" stands for "partial bubble sort."

```
          procedure testpblsort is

            SS: constant integer := 100;

            type ARRAY_TYPE is array (0..SS-1) of integer;
            X :  ARRAY_TYPE;

            procedure pblsort(N,S : in integer) is
            i : integer;
            temp : integer;
            begin
                    for j in 0..N-1 loop
                            i := j;
                            while i < S-1 loop
                                    i := i+1;
                                    if X(j) > X(i) then
                                            temp := X(j);
                                            X(j) := X(i);
                                            X(i) := temp;
                                    end if;
                            end loop;
                    end loop;
            end pblsort;

    begin
            null;
    end testpblsort;
```

The proof commands and proof trace for this program are thoroughly discussed in Section 3.

A correctness proof for the `ctrsort` program, which uses an array of records, employs exactly the same proof commands as the proof for the simplified version `pblsort`, and can in fact be obtained from the latter by simple systematic changes with a text editor (almost all of these changes simply replace array references of form `x[i]` by references to record fields of form `record(x[i],b)`). The result of doing that for program `ctrsort` is given in the Appendix, together with the corresponding proof-trace.

Proving the correctness of a sorting program is not new to SDVS [1], [2]. What is new here is the use of `for`-loops, which are part of Stage 3 Ada in SDVS 11, but not Stage 2. In part then, this report presents a demonstration of this new feature of Stage 3 Ada. As in most previous cases, the effort of constructing a correctness proof within SDVS has exposed a few bugs in the SDVS code and suggested some enhancements to the system. These too will be discussed in this report.

Section 2 describes the overall conception of the SDVS proof, and Section 3 gives a line-by-line walk-through of the proof. In both sections, we discuss bugs identified and fixed or worked around, as well as possible future enhancements to the system. Section 4 summarizes the status of the proofs, as well as the proposed changes and enhancements to SDVS. The SDVS proof and corresponding proof-trace for the `ctrsort` program are presented in the Appendix.

2 Overview of the Proof

In this section we give an overview and theoretical explanation of the verification for the program `pblsort`, and in Section 3 we give a detailed line-by-line exposition of the proof. As remarked earlier, a proof for `ctrsort`, the version using records, or for the original version with long identifiers, can be obtained from the proof for `pblsort` by making systematic changes with a text editor; the result for `ctrsort` appears in the Appendix.

The proof described here is strongly influenced by issues of convenience and workability within SDVS; in other words, some particular choices of methods are made not because of abstract considerations of elegance or brevity, but because the resulting SDVS proof is shorter or runs more quickly. The explanations for these decisions will be made primarily in the detailed discussion in the next section.

Briefly, we expect the proof to establish these facts:

- The program terminates.
- When it terminates, the array `x` has been sorted; we have the *sorted property*

  ```
  forall p (forall q ( 0 le p & p lt .n & p lt q & q lt .s
                                  --> #x[p] le #x[q])),
  ```

  and the *permutation property*

  `#x`, the final value of `x`, is a permutation of the original value `.x`.

The termination property is proved, in effect, as a side-effect of the symbolic execution technique used by SDVS. That is, symbolic execution is carried out until

```
#testpblsort\pc = exited(testpblsort.pblsort)
```

holds, which implies termination. The permutation property will not be dealt with in this proof, merely for the sake of simplification and to stay within time constraints for publication; we have earlier proven the permutation property for a sorting program [2], and the same techniques would apply equally here. This leaves the sorting property to be proven. SDVS handles single-quantifier statements more conveniently, so we replaced the original double-quantifier form by the following pair:

```
forall p (0 le p & p lt .n - 1 --> #x[p] le #x[p + 1]),
forall q (.n - 1 lt q & q lt .s --> #x[.n - 1] le #x[q]).
```

These are equivalent to the original, and a suitable proof could be given in SDVS.

The program has an outer for-loop with an inner while-loop. Therefore the proof contains corresponding nested inductions. A suitable invariant for the outer induction consists of the two formulas

```
forall p (0 le p & p lt .j - 1 --> .x[p] le .x[p + 1]),
forall q (.j le q & q lt .s --> .x[.j - 1] le .x[q]),
```

A word of explanation as to how loop invariants for these for-loops are chosen is in order. They must meet the following criteria:

- They become true upon substitution of the starting value of the for-loop index .j, namely 0, for .j.
- The desired results are provable from them upon substitution of one more than the final value of the for-loop index, namely .n, for .j.
- They contain enough information about the state of the program to permit a proof by induction.

The present case is typical in that the first two of these conditions are acheived merely by substituting .n in the desired results by the loop index, .j. It is atypical in that the situation is sufficiently simple that the third condition is satisfied already by the formulas resulting from that substitution. A more usual circumstance would lead us to seek additional formulas giving more detail about the program environment in order to make the complete proof possible. Most of the creative effort involved in devising loop invariants would normally be concerned with these additional formulas; but in this case none are required.

For the inner induction, we use the same two formulas from the invariant of the outer induction, plus one more:

```
forall p (0 le p & p lt .j - 1 --> .x[p] le .x[p + 1]),
forall q (.j le q & q lt .s --> .x[.j - 1] le .x[q]),
forall q (.j lt q & q le .i --> .x[.j] le .x[q]).
```

With this choice for the inner loop invariants, however, it is necessary to make a case distinction on whether .j = 0 at several places in the proof for the step case of the inner induction (because one needs to avoid references to .x[.j - 1] with .j = 0 as part of the invocation of some axioms). It seemed preferable to avoid this by symbolically executing through the outer loop once for .j = 0, using as an invariant for the inner loop induction

```
forall q (0 lt q & q le .i --> .x[0] le .x[q]),
```

and then to do the nested inductions with invariants as above, for the values of j > 0.

The proof begins with symbolic execution into the for-loop, up to the while-statement. At this point, .j = 0. Then an induction on .i is entered, for .i from 0 to .s - 1 (the upper limit of the array). The invariant of this induction is

```
forall q (0 lt q & q le .i --> .x[0] le .x[q]).
```

This is trivially true when `.i = 0`, and the result of the induction is

```
forall q (0 lt q & q le .s - 1 --> .x[0] le .x[q]).
```

The details of proving this result are given in the next section.

At this point, symbolic execution is once more at the beginning of the `for`-loop. An induction on `.j` from `1` to `.n` is entered, with the invariants

```
forall p (0 le p & p lt .j - 1 --> .x[p] le .x[p + 1]),
forall q (.j le q & q lt .s --> .x[.j - 1] le .x[q]).
```

When `.j = 1`, the first of these is trivial and the second is equivalent to the outcome of the earlier `.i` induction.[5]

In the step case, symbolic execution is moved through the assignment to `i` to the beginning of the `while`-loop. Then an induction on `.i` from `.j` to `.s - 1` is entered, with the invariants

```
forall p (0 le p & p lt .j - 1 --> .x[p] le .x[p + 1]),
forall q (.j le q & q lt .s --> .x[.j - 1] le .x[q]),
forall q (.j lt q & q le .i --> .x[.j] le .x[q]).
```

The first two are invariants of the outer induction. The last is trivially true when `.i = .j`. During the symbolic execution for the step case, there is a step of incrementing `.i`, followed by an `if`-statement with a corresponding case distinction in the proof. In the following discussion, let us use the `#` to signify the values of variables at the end of the body of the `while`-loop, and `.` for their values at the beginning. Thus, `#i = .i + 1` and `#j = .j`. At the end of the symbolic execution of the `if`-statement, we have the additional facts

```
#i = .i + 1,
#x[.j] le #x[#i],
forall q (1 le q & q le .s-1 & q ~= .j & ~q = #i --> #x[q] = .x[q]),
.x[.j - 1] le #x[.j],
.x[.j - 1] le #x[#i].
```

These facts, together with the initial invariants, are sufficient to prove the invariants with `#i` replacing `.i` and `#x` replacing `.x`, as required to complete the induction.

The results of the `.i` induction are

```
forall p (0 le p & p lt .j - 1 --> .x[p] le .x[p + 1]),
forall q (.j le q & q lt .s --> .x[.j - 1] le .x[q]),
forall q (.j lt q & q le .s - 1 --> .x[.j] le .x[q]).
```

[5]In this and similar cases, the lower limit of the induction is the same as the first value of the loop index with which the loop will be executed, whereas the upper limit of the induction is one more than the last execution value.

Symbolic execution of the for-loop ends with an assignment #j = .j + 1 generated automatically by the translator. Just as with the discussion of the while-loop, let # be used for final values, and . for initial values. By instantiating the second of the formulas with q = .j, we get .x[.j - 1] le .x[.j], which combines with the first formula to yield

```
forall p (0 le p & p le .j - 1 --> .x[p] le .x[p + 1]).
```

This is equivalent to

```
forall p (0 le p & p lt #j --> .x[p] le .x[p + 1]),
```

the first of the two goals of the .j-induction step case. The third formula above is equivalent to

```
forall q (#j le q & q lt .s --> .x[#j - 1] le .x[q]).
```

which is the second goal of the .j induction.

The final results of the .j induction are

```
forall p (0 le p & p lt .n - 1 --> .x[p] le .x[p + 1]),
forall q (.n le q & q lt .s --> .x[.n - 1] le .x[q]),
```

which are equivalent to the postconditions of the adalemma to be proven.

3 Walk-through of the Proof

The proof trace begins with the usual steps of translating the Ada program, creating an appropriate adalemma, and reading the axioms that will be needed.

```
<sdvs.3> init
   proof name[]: pbl.proof

State Delta Verification System, Version 11

Restricted to authorized users only.

 date -- 7/24/92 16:12:06  Elapsed time is 0 seconds.

 Parsing Stage 3 Ada file -- "~/ctrsort/testpblsort.a"

 Translating Stage 3 Ada file -- "~/ctrsort/testpblsort.a"

 createadalemma -- [sd pre: (.testpblsort\pc = at(testpblsort.pblsort),
                             range(x) = .s,.s gt 0,.n gt 0,.n le .s,
                             origin(x) = 0)
                     comod: (all)
                       mod: (testpblsort\pc,x)
                      post: (forall p (0 le p & p lt .n - 1
                                         --> #x[p] le #x[p + 1]),
                             forall q (.n - 1 lt q & q lt .s
                                         --> #x[.n - 1] le #x[q]),
                             #testpblsort\pc = exited(testpblsort.pblsort))]

 readaxioms "axioms/arraycoverings.axioms"
   -- (pcovering\slice\element,pcovering\slice\slice,pcovering\element,
       pcovering\slice,disjoint\elements,disjoint\slices,
       disjoint\adjacent\slices)

 readaxioms "axioms/origin-arrays.axioms"
   -- (emptyslice,lowerslice,upperslice,totalslice,slicerange,sliceorigin,
       adjacentslices,elementofslice,elementofaconc1,elementofaconc2,
       sliceofaconc)

 date -- 7/24/92 16:12:10  Elapsed time is 4 seconds.
```

The proveadalemma command is next, and results in opening the state delta of the adalemma for proof. The system automatically creates the appropriate environment in which the proof can be carried out. This is accomplished by applying a series of state deltas generated by the translator.

```
 open -- [sd pre: (alldisjoint(testpblsort,.testpblsort),
                   covering(.testpblsort,testpblsort\pc,ss,nn,array_type!first,
                            array_type!last,x,stdin,stdin\ctr,stdout,
                            stdout\ctr),
                   declare(stdout\ctr,type(integer)),
                   declare(stdout,type(polymorphic)),
                   declare(stdin\ctr,type(integer)),
                   declare(stdin,type(polymorphic)),
                declare(x,
                        type(array,0,(0 + range(x)) - 1,type(integer))),
                declare(array_type!last,type(integer)),
                declare(array_type!first,type(integer)),
                declare(nn,type(integer)),declare(ss,type(integer)),
                <adatr pblsort (n, ...);>)
        comod: (all)
          mod: (all)
         post: ([sd pre: (.testpblsort\pc = at(testpblsort.pblsort),
                          range(x) = .s,.s gt 0,.n gt 0,.n le .s,
                          origin(x) = 0)
                  comod: (all)
                    mod: (diff(all,
                               diff(union(testpblsort\pc,ss,nn,
                                          array_type!first,
                                          array_type!last,x,stdin,
                                          stdin\ctr,stdout,stdout\ctr,n,
                                          s),
                                    union(testpblsort\pc,x))))
                   post: (forall p (0 le p & p lt .n - 1
                                    --> #x[p] le #x[p + 1]),
                          forall q (.n - 1 lt q & q lt .s
                                    --> #x[.n - 1] le #x[q]),
                          #testpblsort\pc = exited(testpblsort.pblsort))])]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,testpblsort)
            post: (alldisjoint(testpblsort,.testpblsort,n,s),
                   covering(#testpblsort,.testpblsort,n,s),
                   declare(n,type(integer)),declare(s,type(integer)),
                   <adatr null;>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,n,s)
            post: (#n = .n,#s = .s,
                   <adatr null;>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc)
            post: (#testpblsort\pc = at(testpblsort.pblsort),
                   <adatr null;>)]

go -- breakpoint reached

open -- [sd pre: (.testpblsort\pc = at(testpblsort.pblsort),range(x) = .s,
                  .s gt 0,.n gt 0,.n le .s,origin(x) = 0)
          comod: (all)
            mod: (diff(all,
                       diff(union(testpblsort\pc,ss,nn,array_type!first,
                                  array_type!last,x,stdin,stdin\ctr,
                                  stdout,stdout\ctr,n,s),
                            union(testpblsort\pc,x))))
        post: (forall p (0 le p & p lt .n - 1
                           --> #x[p] le #x[p + 1]),
               forall q (.n - 1 lt q & q lt .s
                           --> #x[.n - 1] le #x[q]),
               #testpblsort\pc = exited(testpblsort.pblsort))]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,testpblsort)
            post: (alldisjoint(testpblsort,.testpblsort,i),
                   covering(#testpblsort,.testpblsort,i),
                   declare(i,type(integer)),
                   <adatr i : integer>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,testpblsort)
            post: (alldisjoint(testpblsort,.testpblsort,temp),
                   covering(#testpblsort,.testpblsort,temp),
                   declare(temp,type(integer)),
                   <adatr temp : integer>)]
```

At this point, the environment of the body of the procedure `pblsort` has been created. The next state delta applied originates from the translation of the `for`-loop at the beginning of the body.

```
apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,testpblsort)
            post: (alldisjoint(testpblsort,.testpblsort,j),
                   covering(#testpblsort,.testpblsort,j),
                   declare(j,type(integer)),
                   <adatr j : constant integer := 0>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,j)
            post: (#j = 0,
                   <adatr j : constant integer := 0>)]

applydecls -- declaration elaboration complete.
```

Ordinarily, one would begin an induction for the outer loop here, when the iteration variable `j` has been declared and initialized. However, it is more convenient in this proof to execute through the loop one time, and then to do an induction for the remaining values of `j`. This is because some of the loop invariants refer to `x[.j - 1]`, and SDVS will enter this expression into its list of places. However, when `.j = 0`, the place referenced is outside the bounds of the array `x`. This does not produce an error, but the relationship of such a place to other places is not determined, and the presence of such an "ambiguous" place may slow down the covering solver. This is not entirely certain, but it seems prudent to avoid the problem. Furthermore, starting the induction here will inevitably result in case distinctions on whether `.j > 0` in other parts of the proof; in particular, some of the axioms will not allow references to `x[.j - 1]` when `.j = 0`, so case distinctions would be needed for invocations of those axioms. Dealing with the case .j = 0 separately makes a small but significant reduction in the overall complexity of our proof.

```
apply -- [sd pre: (.j le .n - 1)
           comod: (all)
             mod: (testpblsort\pc)
            post: (<adatr for ...>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,i)
            post: (#i = .j,
                   <adatr i := j;>)]
```

At this point, symbolic execution has reached the `while`-loop. We will begin an induction on `.i`. The induction is from 0, which is the value of `.i` upon initial entry to the body of the loop, to `.s - 1`, which is one more than the value of `.i` upon the very last entry to the body of the loop. With a `while`-loop structured as this one is, the upper limit of the induction is the same as the value used in the criterion for exiting the loop.

The invariant of the induction, primarily the formula

```
forall q (0 lt q & q le .i --> .x[0] le .x[q]),
```

is chosen so that

- it becomes true upon substitution of the initial value 0 for `.i`,
- the substitution of the final value `.s - 1` for `.i` results in a formula that implies the desired formula

  ```
  forall q (0 lt q & q lt .s --> #x[0] le #x[q]).
  ```

  This desired formula is determined by consideration of the needs of subsequent parts of the proof; in this case establishing the initial condition of the induction on `.j` from 1 to `.n`, which appears later in the proof. That in turn will be determined by examining the postcondition of the adalemma to be proven. In other words, one must look ahead to determine which steps to take at this point; the precise role of these formulas will become clear as the proof progresses.

Essentially the same strategy is applied to determine the bounds and invariants of all the other inductions appearing in our proof.

Before beginning the induction, we give names to the two usable state deltas which represent the alternative continuations possible from the `while` statement: one (`u(1)` for the case that the condition for entry into the body of the loop is not satisfied, the other (`u(2)`) for the case that it is satisfied. These names are needed to conveniently express the invariant of the induction. We also give a name to the current universe of places, which is needed to express the comod list and the postcondition for the induction.

```
letsd -- j0.i.loop.exit.sd = u(1)
```

```
letsd -- j0.i.loop.step.sd = u(2)

let -- j0.i.induct.universe = .testpblsort

induction -- .i from 0 to .s - 1

  open -- [sd pre: (true)
            comod: (all)
             post: ([sd pre: (~(.i lt .s - 1))
                      comod: (all)
                        mod: (testpblsort\pc)
                       post: (<adatr while i < s - 1

                                       i := i + 1;
                                       ...
                                     end loop;>)],
                    [sd pre: (.i lt .s - 1)
                      comod: (all)
                        mod: (testpblsort\pc)
                       post: (<adatr while i < s - 1

                                       i := i + 1;
                                       ...
                                     end loop;>)],
                    forall q (0 lt q & q le .i --> .x[0] le .x[q]),
                    covering(j0.i.induct.universe,.testpblsort),.i = 0)]

  close -- 0 steps/applications

  open -- [sd pre: (.i ge 0,.i lt .s - 1,
                    [sd pre: (~(.i lt .s - 1))
                      comod: (all)
                        mod: (testpblsort\pc)
                       post: (<adatr while i < s - 1

                                       i := i + 1;
                                       ...
                                     end loop;>)],
                    [sd pre: (.i lt .s - 1)
                      comod: (all)
                        mod: (testpblsort\pc)
                       post: (<adatr while i < s - 1

                                       i := i + 1;
                                       ...
                                     end loop;>)],
                    forall q (0 lt q & q le .i --> .x[0] le .x[q]),
                    covering(j0.i.induct.universe,.testpblsort))
            comod: (diff(j0.i.induct.universe,
                         union(testpblsort\pc,i,temp,x[0:(.s - 1)])))
              mod: (testpblsort\pc,i,temp,x[0:(.s - 1)])
             post: ([sd pre: (~(.i lt .s - 1))
                      comod: (all)
                        mod: (testpblsort\pc)
                       post: (<adatr while i < s - 1
                                    i := i + 1;
                                    ...
                                  end loop;>)],
                  [sd pre: (.i lt .s - 1)
                    comod: (all)
                      mod: (testpblsort\pc)
                     post: (<adatr while i < s - 1

                                    i := i + 1;
                                    ...
                                  end loop;>)],
                  forall q (0 lt q & q le #i --> #x[0] le #x[q]),
                  covering(j0.i.induct.universe,#testpblsort),
                  #i = .i + 1)]

apply -- [sd pre: (.i lt .s - 1)
           comod: (all)
             mod: (testpblsort\pc)
            post: (<adatr while i < s - 1

                           i := i + 1;
                           ...
                         end loop;>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,i)
            post: (#i = .i + 1,
                   <adatr i := i + 1;>)]

cases -- .x[0] gt .x[.i]

  open -- [sd pre: (.x[0] gt .x[.i])
            comod: (all)
              mod: (testpblsort\pc,i,temp,x[0:(.s - 1)])
             post: ([sd pre: (~(.i lt .s - 1))
                      comod: (all)
                        mod: (testpblsort\pc)
                       post: (<adatr while i < s - 1


                                      i := i + 1;
                                      ...
                                    end loop;>)],
                    [sd pre: (.i lt .s - 1)
                      comod: (all)
                        mod: (testpblsort\pc)
                       post: (<adatr while i < s - 1

                                      i := i + 1;
                                      ...
                                    end loop;>)],
                    forall q (0 lt q & q le #i --> #x[0] le #x[q]),
                    covering(j0.i.induct.universe,#testpblsort),
                         #i = i\51 + 1)]

apply -- [sd pre: (.x[.j] gt .x[.i],.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1,
                   .i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
          comod: (all)
            mod: (testpblsort\pc)
           post: (<adatr if x (j) > x (i)
                                 temp := x (j);
                                 ...
                             end if;>)]

apply -- [sd pre: (.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1)
          comod: (all)
            mod: (testpblsort\pc,temp)
           post: (#temp = .x[.j],
                  <adatr temp := x (j);>)]
```

At this point, the following is true, and can easily by proved:

```
forall q (0

lt q & q lt .i --> .temp le .x[q])
```

The next program statement for symbolic execution is

```
x(j) := x(i);
```

However, it turns out that the application of the next translated state delta would cause the deletion of the quantified statement from the usablequantifiers list, even though `x[.j]` is obviously disjoint from the part of `x`, namely `x[1:.i-1]`, on which the quantified statement depends. This is an artifact of the particular heuristics currently used by SDVS to determine which quantified formulas may be retained after the application of a state delta. They result in criteria which are, in this case at least, overly restrictive.

Let us define the *support* of a quantified formula as the smallest set of places upon which the truth of the formula depends—or, to put it another way, such that the truth of the formula is independent of the values of places not in the set. The existing heuristics try to approximate the support of quantified formulas. They do this in a very conservative way which may result in the deletion of formulas that could be retained (as would be the case here were we to proceed in the normal way).

Instead, we have a work-around: prove the formula, assert the appropriate disjointness condition, then prove a static state delta with only the desired quantified formula as its postcondition. The comod list of this state delta should be the precise support of the formula. Such a state delta will not be deleted by the hueristics; the test for retention of state deltas is a simple test for disjointness of the comod list of the state delta to be retained and the mod list of the state delta being applied. Then after the adatr state delta is applied, the static state delta can be applied to regain the desired formula.

(We are still taking advantage of a quirk in SDVS' handling of quantified formulas. When the proof of the static state delta is open, its postcondition is found among the usablequantifiers, so the proof closes immediately. The tests for retention of formulas in this situation of a new proof context differ in subtle but important ways from those involved with application of an state delta. The entire situation should probably be considered to indicate a bug in SDVS. It is important to note, however, that the bug does not invalidate proofs—SDVS is erring on the conservative side by deleting formulas that could be retained, as opposed to retaining ones that should be deleted.)

```
provebygeneralization -- forall q (0 lt q &
                                   q lt 1 + i\51
                                   --> .x[0] le .x[q])

provebyaxiom disjoint\slices -- alldisjoint(x[1:(.i - 1)],
                                            x[0:0])

open -- [sd pre: (true)
          comod: (temp,i,x[1:(.i - 1)])
           post: (forall q (0 lt q & q lt .i --> .temp le .x[q]))]

close -- 0 steps/applications

apply -- [sd pre: (.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1,
                   .i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testpblsort\pc,x[.j])
            post: (#x[.j] = .x[.i],
                   <adatr x (j) := x (i);>)]

apply -- [sd pre: (true)
           comod: (temp,i,x[1:((1 + i\51) - 1)])
            post: (forall q (0 lt q & q lt .i
                             --> .temp le .x[q]))]
```

As expected, the static state delta remained usable past the application of the adatr state delta for the `x(j) := x(i);` assignment (which was the penultimate application above), so its application (the last application above) restored the quantifier to the usablequantifiers list. This same technique will be used without further comment in several places below.

```
provebyaxiom disjoint\slices -- alldisjoint(x[1:(.i - 1)],
                                            x[.i:.i])

apply -- [sd pre: (.i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testpblsort\pc,x[.i])
            post: (#x[.i] = .temp,
                   <adatr x (i) := temp;>)]

apply -- [sd pre: (true)
           comod: (temp,i,x[1:((1 + i\51) - 1)])
            post: (forall q (0 lt q & q lt .i
                             --> .temp le .x[q]))]
```

```
  provebygeneralization -- forall q (0 lt q &
                                     q le 1 + i\51
                                     --> .x[0] le .x[q])

 close -- 11 steps/applications

 open -- [sd pre: (~(.x[0] gt .x[.i]))
          comod: (all)
            mod: (testpblsort\pc,i,temp,x[0:(.s - 1)])
           post: ([sd pre: (~(.i lt .s - 1))
                    comod: (all)
                      mod: (testpblsort\pc)
                     post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                    end loop;>)],
                  [sd pre: (.i lt .s - 1)
                    comod: (all)
                      mod: (testpblsort\pc)
                     post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                    end loop;>)],
                  forall q (0 lt q & q le #i --> #x[0] le #x[q]),
                  covering(j0.i.induct.universe,#testpblsort),
                  #i = i\51 + 1)]

  apply -- [sd pre: (~(.x[.j] gt .x[.i]),.j ge origin(x),
                     .j le (origin(x) + range(x)) - 1,
                     .i ge origin(x),
                     .i le (origin(x) + range(x)) - 1)
             comod: (all)
               mod: (testpblsort\pc)
              post: (<adatr if x (j) > x (i)
                              temp := x (j);
                                 ...
                            end if;>)]

  provebygeneralization -- forall q (0 lt q &
                                     q le 1 + i\51
                                     --> .x[0] le .x[q])

 close -- 2 steps/applications

join -- [sd pre: (true)
         comod: (all)
           mod: (testpblsort\pc,i,temp,x[0:(.s - 1)])
          post: ([sd pre: (~(.i lt .s - 1))
                   comod: (all)
                     mod: (testpblsort\pc)
                    post: (<adatr while i < s - 1
                                        i := i + 1;
                                        ...
                                      end loop;>)],
                          [sd pre: (.i lt .s - 1)
                            comod: (all)
                              mod: (testpblsort\pc)
                             post: (<adatr while i < s - 1

                                        i := i + 1;
                                        ...
                                      end loop;>)],
                          forall q (0 lt q & q le #i --> #x[0] le #x[q]),
                          covering(j0.i.induct.universe,#testpblsort),
                          #i = i\51 + 1)]

      inserting -- pcovering(all,x[0:(.s - 1)])

      inserting -- pcovering(all,x[0:(.s - 1)])

    close -- 3 steps/applications

  join induction cases -- [sd pre: (0 le .s - 1)
                            comod: (all,
                                    diff(j0.i.induct.universe,
                                         union(testpblsort\pc,i,temp,
                                               x[0:(.s - 1)])))
                              mod: (testpblsort\pc,i,temp,x[0:(.s - 1)])
                             post: (#i = .s - 1,
                                    formula(j0.i.loop.exit.sd),
                                    formula(j0.i.loop.step.sd),
                                    forall q (0 lt q & q le #i
                                                --> #x[0] le #x[q]),
                                    covering(j0.i.induct.universe,
                                             #testpblsort))]
```

The induction for the `while`-loop is completed. Symbolic execution must now be advanced back to the beginning of the `for`-loop.

```
  inserting -- pcovering(all,x[0:(.s - 1)])

  inserting -- pcovering(all,x[0:(.s - 1)])

  apply -- [sd pre: (~(.i lt .s - 1))
             comod: (all)
               mod: (testpblsort\pc)
              post: (<adatr while i < s - 1

                           i := i + 1;
                           ...
                         end loop;>)]

  apply -- [sd pre: (true)
             comod: (all)
               mod: (testpblsort\pc,j)
              post: (#j = .j + 1,
                     <adatr j := j + 1;>)]
```

The last state delta applied was generated by the translator as part of the elaboration of the for-loop.

Symbolic execution has here returned to the beginning of the for-loop. From here the proof proceeds with two nested inductions, one for the outer for-loop over values of j > 0, and the other for the inner while-loop over values of i from j through S - 1. The bounds and invariants for the inductions are obtained by following the procedure outlined earlier. Thus, the lower bound for the .j induction is 1 (the current value of j and also the first value for which the loop body would be executed in subsequent steps), and the upper bound is .n (which is one more than the last value of j for which the loop is executed). The invariant formulas

```
forall p (0 le p & p lt .j - 1 --> .x[p] le .x[p + 1]),

forall q (.j le q & q lt .s --> .x[.j - 1] le .x[q]),
```

are developed from the postconditions of the adalemma to be proven, and are chosen to be true when .j = 1 (as now) and to imply the adalemma postconditions when .j = .n.

```
letsd -- j.loop.exit.sd = u(1)

letsd -- j.loop.step.sd = u(2)

let -- j.induct.universe = .testpblsort

induction -- .j from 1 to .n

  open -- [sd pre: (true)
           comod: (all)
            post: ([sd pre: (.j gt .n - 1)
                     comod: (all)
                       mod: (testpblsort\pc)
                      post: (<adatr for ...>)],
                   [sd pre: (.j le .n - 1)
                     comod: (all)
                       mod: (testpblsort\pc)
                      post: (<adatr for ...>)],
                   forall p (0 le p & p lt .j - 1
                               --> .x[p] le .x[p + 1]),
                   forall q (.j le q & q lt .s
                               --> .x[.j - 1] le .x[q]),
                   covering(j.induct.universe,.testpblsort),.j = 1)]

    provebygeneralization -- forall q (1 le q & q lt .s
                                          --> .x[0] le .x[q])

  close -- 1 steps/applications

  open -- [sd pre: (.j ge 1,.j lt .n,
                   [sd pre: (.j gt .n - 1)
                     comod: (all)
                       mod: (testpblsort\pc)
                      post: (<adatr for ...>)],
                   [sd pre: (.j le .n - 1)
```

```
                        comod: (all)
                          mod: (testpblsort\pc)
                         post: (<adatr for ...>)],
                     forall p (0 le p & p lt .j - 1
                                 --> .x[p] le .x[p + 1]),
                     forall q (.j le q & q lt .s
                                 --> .x[.j - 1] le .x[q]),
                     covering(j.induct.universe,.testpblsort))
         comod: (diff(j.induct.universe,
                           union(j,i,x,temp,testpblsort\pc)))
           mod: (j,i,x,temp,testpblsort\pc)
          post: ([sd pre: (.j gt .n - 1)
                        comod: (all)
                          mod: (testpblsort\pc)
                         post: (<adatr for ...>)],
                     [sd pre: (.j le .n - 1)
                        comod: (all)
                          mod: (testpblsort\pc)
                         post: (<adatr for ...>)],
                     forall p (0 le p & p lt #j - 1
                                 --> #x[p] le #x[p + 1]),
                     forall q (#j le q & q lt #s
                                 --> #x[#j - 1] le #x[q]),
                     covering(j.induct.universe,#testpblsort),
                     #j = .j + 1)]

inserting -- pcovering(x,x[.j - 1])

apply -- [sd pre: (.j le .n - 1)
           comod: (all)
             mod: (testpblsort\pc)
            post: (<adatr for ...>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,i)
            post: (#i = .j,
                   <adatr i := j;>)]
```

Now we begin the inner induction on .i, for the `while`-loop. The bounds are chosen according the same criteria as before. The invariants are developed by examining the post-conditions of the state delta for the outer induction, just as the outer induction invariants were determined by examining the postconditions of the adalemma state delta.

```
letsd -- i.loop.exit.sd = u(1)

letsd -- i.loop.step.sd = u(2)

let -- i.induct.universe = .testpblsort

induction -- .i from .j to .s - 1

  open -- [sd pre: (true)
           comod: (all)
```

```
          post: ([sd pre: (~(.i lt .s - 1))
                   comod: (all)
                     mod: (testpblsort\pc)
                    post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                   end loop;>)],
                 [sd pre: (.i lt .s - 1)
                   comod: (all)
                     mod: (testpblsort\pc)
                    post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                   end loop;>)],
                 forall p (0 le p & p lt .j - 1
                             --> .x[p] le .x[p + 1]),
                 forall q (.j le q & q lt .s
                             --> .x[.j - 1] le .x[q]),
                 forall q (.j lt q & q le .i --> .x[.j] le .x[q]),
                 covering(i.induct.universe,.testpblsort),
                 .i = .j)]

close -- 0 steps/applications

open -- [sd pre: (.i ge .j,.i lt .s - 1,
                 [sd pre: (~(.i lt .s - 1))
                   comod: (all)
                     mod: (testpblsort\pc)
                    post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                   end loop;>)],
                 [sd pre: (.i lt .s - 1)
                   comod: (all)
                     mod: (testpblsort\pc)
                    post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                   end loop;>)],
                 forall p (0 le p & p lt .j - 1
                             --> .x[p] le .x[p + 1]),
                 forall q (.j le q & q lt .s
                             --> .x[.j - 1] le .x[q]),
                 forall q (.j lt q & q le .i --> .x[.j] le .x[q]),
                 covering(i.induct.universe,.testpblsort))
         comod: (diff(i.induct.universe,
                      union(testpblsort\pc,i,temp,
                            x[.j:(.s - 1)])))
           mod: (testpblsort\pc,i,temp,x[.j:(.s - 1)])
          post: ([sd pre: (~(.i lt .s - 1))
                   comod: (all)
                 mod: (testpblsort\pc)
                post: (<adatr while i < s - 1

                                i := i + 1;
                                ...
                              end loop;>)],
             [sd pre: (.i lt .s - 1)
               comod: (all)
                 mod: (testpblsort\pc)
                post: (<adatr while i < s - 1

                                i := i + 1;
                                ...
                              end loop;>)],
             forall p (0 le p & p lt #j - 1
                         --> #x[p] le #x[p + 1]),
             forall q (#j le q & q lt #s
                         --> #x[#j - 1] le #x[q]),
             forall q (#j lt q & q le #i --> #x[#j] le #x[q]),
             covering(i.induct.universe,#testpblsort),
             #i = .i + 1)]

inserting -- pcovering(x,x[.j - 1])

apply -- [sd pre: (.i lt .s - 1)
           comod: (all)
             mod: (testpblsort\pc)
            post: (<adatr while i < s - 1

                            i := i + 1;
                            ...
                          end loop;>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,i)
            post: (#i = .i + 1,
                   <adatr i := i + 1;>)]

cases -- .x[.j] gt .x[.i]

  open -- [sd pre: (.x[.j] gt .x[.i])
            comod: (all)
              mod: (testpblsort\pc,i,temp,x[.j:(.s - 1)])
             post: ([sd pre: (~(.i lt .s - 1))
                      comod: (all)
                        mod: (testpblsort\pc)
                       post: (<adatr while i < s - 1

                                       i := i + 1;
                                       ...
                                     end loop;>)],
                    [sd pre: (.i lt .s - 1)
                      comod: (all)
                        mod: (testpblsort\pc)
              post: (<adatr while i < s - 1
                                    i := i + 1;
                                    ...
                                  end loop;>)],
            forall p (0 le p & p lt #j - 1
                         --> #x[p] le #x[p + 1]),
            forall q (#j le q & q lt #s
                         --> #x[#j - 1] le #x[q]),
            forall q (#j lt q & q le #i
                         --> #x[#j] le #x[q]),
            covering(i.induct.universe,#testpblsort),
            #i = i\143 + 1)]

apply -- [sd pre: (.x[.j] gt .x[.i],.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1,
                   .i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testpblsort\pc)
            post: (<adatr if x (j) > x (i)
                               temp := x (j);
                                     ...
                           end if;>)]

apply -- [sd pre: (.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testpblsort\pc,temp)
            post: (#temp = .x[.j],
                   <adatr temp := x (j);>)]

provebyinstantiation -- .x[.j - 1] le .x[.j]

provebyinstantiation -- .x[.j - 1] le .x[1 + i\143]

provebyaxiom disjoint\slices -- alldisjoint(x[(.i + 1)
                                              :(.s - 1)],
                                            x[.j:.j])

provebyaxiom disjoint\slices -- alldisjoint(x[(.j + 1)
                                              :(.i - 1)],
                                            x[.j:.j])

provebyaxiom disjoint\slices -- alldisjoint(x[0
                                              :(.j - 1)],
                                            x[.j:.j])

provebyaxiom pcovering\slice\element -- pcovering(x[0
                                                    :(.j - 1)],
                                                  x[.j - 1])

provebygeneralization -- forall q ((.j lt q &
                                    q lt .s) &
                                  q ~= 1 + i\143
```

```
                                     --> .x[.j - 1] le .x[q])

provebygeneralization -- forall q (.j lt q &
                                     q lt 1 + i\143
                                     --> .x[.j] le .x[q])

open -- [sd pre: (true)
          comod: (j,s,i,temp,x[0:(.j - 1)],
                  x[(.j + 1):(.i - 1)],
                  x[(.i + 1):(.s - 1)])
           post: (forall p (0 le p & p lt .j - 1
                              --> .x[p] le .x[p + 1]),
                  forall q ((.j lt q & q lt .s) &
                              q ~= .i
                              --> .x[.j - 1] le .x[q]),
                  forall q (.j lt q & q lt .i
                              --> .temp le .x[q]))]

close -- 0 steps/applications

provebyaxiom pcovering\slice\element -- pcovering(x[.j
                                                     :(.s - 1)],
                                                   x[.j])

apply -- [sd pre: (.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1,
                   .i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
          comod: (all)
            mod: (testpblsort\pc,x[.j])
           post: (#x[.j] = .x[.i],
                  <adatr x (j) := x (i);>)]

provebyaxiom pcovering\slice\element -- pcovering(x[.j
                                                     :(.s - 1)],
                                                   x[.i])

provebyaxiom disjoint\slices -- alldisjoint(x[0
                                              :(.j - 1)],
                                            x[.i:.i])

provebyaxiom disjoint\slices -- alldisjoint(x[(.i + 1)
                                              :(.s - 1)],
                                            x[.i:.i])

provebyaxiom disjoint\slices -- alldisjoint(x[(.j + 1)
                                              :(.i - 1)],
                                            x[.i:.i])

apply -- [sd pre: (.i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
          comod: (all)
            mod: (testpblsort\pc,x[.i])
           post: (#x[.i] = .temp,
                  <adatr x (i) := temp;>)]
```

```
  apply -- [sd pre: (true)
           comod: (j,s,i,temp,x[0:(.j - 1)],
                  x[(.j + 1)
                    :((1 + i\143) - 1)],
                  x[((1 + i\143) + 1):(.s - 1)])
            post: (forall p (0 le p &
                              p lt .j - 1
                              --> .x[p] le .x[p + 1]),
                  forall q ((.j lt q & q lt .s) &
                              q ~= .i
                              --> .x[.j - 1] le .x[q]),
                  forall q (.j lt q & q lt .i
                              --> .temp le .x[q]))]

  provebygeneralization -- forall q (.j lt q &
                                    q le 1 + i\143
                                    --> .x[.j] le .x[q])

  provebygeneralization -- forall q (.j le q & q lt .s
                                    --> .x[.j - 1] le .x[q])

close -- 21 steps/applications

open -- [sd pre: (~(.x[.j] gt .x[.i]))
        comod: (all)
          mod: (testpblsort\pc,i,temp,x[.j:(.s - 1)])
         post: ([sd pre: (~(.i lt .s - 1))
                  comod: (all)
                    mod: (testpblsort\pc)
                   post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                   end loop;>)],
               [sd pre: (.i lt .s - 1)
                  comod: (all)
                    mod: (testpblsort\pc)
                   post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                   end loop;>)],
               forall p (0 le p & p lt #j - 1
                           --> #x[p] le #x[p + 1]),
               forall q (#j le q & q lt #s
                           --> #x[#j - 1] le #x[q]),
               forall q (#j lt q & q le #i
                           --> #x[#j] le #x[q]),
               covering(i.induct.universe,#testpblsort),
               #i = i\143 + 1)]

  apply -- [sd pre: (~(.x[.j] gt .x[.i]),.j ge origin(x),
                  .j le (origin(x) + range(x)) - 1,
                  .i ge origin(x),
                            .i le (origin(x) + range(x)) - 1)
                    comod: (all)
                      mod: (testpblsort\pc)
                     post: (<adatr if x (j) > x (i)
                                     temp := x (j);
                                     ...
                                 end if;>)]

        go -- no more declarations or statements

        provebygeneralization -- forall q (.j lt q &
                                           q le 1 + i\143
                                           --> .x[.j] le .x[q])

      close -- 2 steps/applications

    join -- [sd pre: (true)
              comod: (all)
                mod: (testpblsort\pc,i,temp,x[.j:(.s - 1)])
               post: ([sd pre: (~(.i lt .s - 1))
                        comod: (all)
                          mod: (testpblsort\pc)
                         post: (<adatr while i < s - 1

                                         i := i + 1;
                                         ...
                                       end loop;>)],
                      [sd pre: (.i lt .s - 1)
                        comod: (all)
                          mod: (testpblsort\pc)
                         post: (<adatr while i < s - 1

                                         i := i + 1;
                                         ...
                                       end loop;>)],
                      forall p (0 le p & p lt #j - 1
                                --> #x[p] le #x[p + 1]),
                      forall q (#j le q & q lt #s
                                --> #x[#j - 1] le #x[q]),
                      forall q (#j lt q & q le #i
                                --> #x[#j] le #x[q]),
                      covering(i.induct.universe,#testpblsort),
                      #i = i\143 + 1)]

    inserting -- pcovering(all,x[.j:(.s - 1)])

    inserting -- pcovering(x,x[.j - 1])

    inserting -- pcovering(all,x[.j:(.s - 1)])

  close -- 3 steps/applications

join induction cases -- [sd pre: (.j le .s - 1)
                         comod: (all,
                                 diff(i.induct.universe,
                                              union(testpblsort\pc,i,temp,
                                                   x[.j
                                                     :(.s - 1)]))
                                    mod: (testpblsort\pc,i,temp,
                                          x[.j:(.s - 1)])
                                   post: (#i = .s - 1,
                                          formula(i.loop.exit.sd),
                                          formula(i.loop.step.sd),
                                          forall p (0 le p &
                                                    p lt #j - 1
                                                    --> #x[p]
                                                          le #x[p + 1]),
                                          forall q (#j le q & q lt #s
                                                    --> #x[#j - 1] le #x[q]),
                                          forall q (#j lt q & q le #i
                                                    --> #x[#j] le #x[q]),
                                          covering(i.induct.universe,
                                                  #testpblsort))]

inserting -- pcovering(all,x[.j:(.s - 1)])

inserting -- pcovering(x,x[.j - 1])

inserting -- pcovering(all,x[.j:(.s - 1)])

apply -- [sd pre: (~(.i lt .s - 1))
           comod: (all)
             mod: (testpblsort\pc)
            post: (<adatr while i < s - 1

                             i := i + 1;
                             ...
                           end loop;>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testpblsort\pc,j)
            post: (#j = .j + 1,
                   <adatr j := j + 1;>)]

go -- no more declarations or statements

subcases -- .j - 1 gt 0

  open -- [sd pre: (.j - 1 gt 0)
            comod: (all)
             post: (forall p (0 le p & p lt .j - 1
                                --> .x[p] le .x[p + 1]))]

    provebyinstantiation -- .x[j\127 - 1] le .x[j\127]

    provebygeneralization -- forall p (0 le p &
                                       p  lt (1 + j\127) - 1
                                       --> .x[p] le .x[p + 1])
            close -- 2 steps/applications

            open -- [sd pre: (~(.j - 1 gt 0))
                      comod: (all)
                       post: (forall p (0 le p & p lt .j - 1
                                        --> .x[p] le .x[p + 1]))]

              The state delta is vacuously TRUE because its precondition is
              FALSE.

            close -- 0 steps/applications

          join -- [sd pre: (true)
                    comod: (all)
                     post: (forall p (0 le p & p lt .j - 1
                                      --> .x[p] le .x[p + 1]))]

          provebygeneralization -- forall q (1 + j\127 le q & q lt .s
                                            --> .x[j\127] le .x[q])

        close -- 10 steps/applications

      join induction cases -- [sd pre: (1 le .n)
                                comod: (all,
                                        diff(j.induct.universe,
                                             union(j,i,x,temp,testpblsort\pc)))
                                  mod: (j,i,x,temp,testpblsort\pc)
                                 post: (#j = .n,formula(j.loop.exit.sd),
                                        formula(j.loop.step.sd),
                                        forall p (0 le p &
                                                  p lt #j - 1
                                                  --> #x[p] le #x[p + 1]),
                                        forall q (#j le q & q lt #s
                                                  --> #x[#j - 1] le #x[q]),
                                        covering(j.induct.universe,#testpblsort))]

      inserting -- pcovering(x,x[.n - 1])
```

Both inductions are completed, and this brings symbolic execution to the end of the body of the `pblsort` procedure. The remaining proof step is to use the result of the `.j`-induction to derive the precise required form of the postconditions of the adalemma state delta, which differs slightly from the formulas proved by the induction (using `.n - 1 lt q` instead of `.n le q`.) After that, the remaining steps of symbolic execution undeclare the local environment of the procedure and fulfill the termination condition.

```
      provebygeneralization -- forall q (.n - 1 lt q & q lt .s
                                        --> .x[.n - 1] le .x[q])

      apply -- [sd pre: (.j gt .n - 1)
                 comod: (all)
                   mod: (testpblsort\pc)
                  post: (<adatr for ...>)]

      apply -- [sd pre: (true)
```

```
                comod: (all)
                  mod: (testpblsort\pc,testpblsort,j)
                 post: (covering(.testpblsort,#testpblsort,j),undeclare(j),
                        <adatr j : constant integer := 0>)]

    apply -- [sd pre: (true)
                comod: (all)
                  mod: (testpblsort\pc,testpblsort,temp)
                 post: (covering(.testpblsort,#testpblsort,temp),undeclare(temp),
                        <adatr temp : integer>)]

    apply -- [sd pre: (true)
                comod: (all)
                  mod: (testpblsort\pc,testpblsort,i)
                 post: (covering(.testpblsort,#testpblsort,i),undeclare(i),
                        <adatr i : integer>)]

    apply -- [sd pre: (true)
                comod: (all)
                  mod: (testpblsort\pc)
                 post: (#testpblsort\pc = exited(testpblsort.pblsort),
                        <adatr null;>)]

  close -- 22 steps/applications

close -- 4 steps/applications

proveadalemma -- [sd pre: (.testpblsort\pc = at(testpblsort.pblsort),
                           range(x) = .s,.s gt 0,.n gt 0,.n le .s,
                           origin(x) = 0)
                    comod: (all)
                      mod: (testpblsort\pc,x)
                     post: (forall p (0 le p & p lt .n - 1
                                        --> #x[p] le #x[p + 1]),
                           forall q (.n - 1 lt q & q lt .s
                                        --> #x[.n - 1] le #x[q]),
                           #testpblsort\pc = exited(testpblsort.pblsort))]

date -- 7/24/92 16:19:44  Elapsed time is 7 minutes and 34 seconds.
```

4 Conclusions

We have demonstrated SDVS' ability to accomplish correctness proofs for at least some operational programs. SDVS has also been enhanced to handle `for`-loops, a step that should be important in applications, since `for`-loops, even though less general, are probably used more often than `while`-loops.

A second question posed by our original source was whether it could be proved that the program did not access locations outside the boundary of the array. A partial answer to this question is provided by the SDVS proof. Since the mod list of the adalemma proved includes only places within the array, we know that the program does not *alter* any locations outside the boundary of the array. The SDVS proof leaves open the question of whether any read-access was made to a location outside the boundary; indeed, the present configuration of SDVS does not provide a means of answering such a question. However, an SDVS proof can provide information relevant to the question.

Let us consider the question, "Why might one want to know whether a program accesses locations outside the bounds of the array?" By asking our original source, we learned that the main reason (from his point of view) was to enable turning off run-time in-bounds checks for array references. It seems that in some programs, including this one, these run-time array-reference checks can consume as much as half the processing time. A proof that a program cannot reference out-of-bounds would justify turning off the checks and thereby improve execution-time significantly.

There may be several reasons why it is necessary to prevent out-of-bounds array references, among them

1. to protect data outside the array from alteration
2. to avoid unwanted hardware effects
3. to meet data security needs

Memory accesses in a computer are usually distinguished as either *writes* or *reads.* Item (1) concerns out-of-bounds writes. SDVS deals with these effectively: correctness statements about programs are formulated as state deltas, and proving such a state delta implies the assurance that the program alters no data outside the mod list. Thus, one need only limit the mod list to in-bounds places and prove the sd to obtain a proof that the program makes no out-of-bounds writes.

The sort of hardware effects we are referring to in item (2) are illustrated by a common computer-design "trick" for machines in which the hardware is memory-mapped: use high-order bits on the address bus, which are not needed for actual memory accesses, to control hardware functions. For example, on a machine with a 32-bit address bus but no memory locations requiring bits above 27 for addressing, the designer might use bit 28 to select/deselect the disk drive. So even an out-of-bounds read can have a hardware effect. (This situation is quite common on personal computers.) The current version of SDVS (Version 11) has no ability to detect such effects.

In the last item, the essential point is not so much whether a "forbidden" read-access occurred, but whether information obtained through such an access could be communicated by the program through its output. This issue becomes important, for example, in the case of some operating systems where a user-supplied program may be executed with a privileged status, allowing it access to data not normally available directly to the user. SDVS has some ability to deal with this type of question. Our present proof, when supplemented by the proof that the final array is a permutation of the original, would rule out most such surreptitious communications.

On the basis of our present proof, we know only that the first `N` elements of the final array are, in sorted order, the least `N` of the original, and the final `S - N` elements are the largest `S - N` elements of the original. Left unspecified is the actual ordering of these `S - N` elements, so it is conceivable that a surreptitious communication could be made with this ordering. For example, assuming `S > N+1`, a binary value could be communicated according to whether or not the last element of the final array was less than its predecessor.

In other words, the adalemma we proved does not provide a full, deterministic specification of the program output in terms of the input, so it leaves open the possibility that the program could communicate additional information through the unspecified aspects of the output. Extending the proof so as to rule out this possibility should be possible by a careful analysis and mathematical statement of the relation between the final unsorted portion of the array and the original.

Of course, in the case of this simple program, one can easily see that no surreptitious communication occurs. The point is whether this fact is indeed captured by the proof we have given. It is not, but we believe that the methodology to improve the proof along these lines is understood.[6]

Some ideas for improving SDVS' handling of quantified formulas have emerged from this effort. Before elaborating upon them, we review the existing mechanisms for handling quantifiers.

At any point in symbolic execution, SDVS maintains a list of "usablequantifiers," which are quantified formulas known to be true at that time. A full range of logical facilities for using and proving such formulas is available from EKL, the quantification-handling subsystem of SDVS. There are also commands that operate independently of EKL and are often more convenient, but that apply only to formulas having leading quantifiers; among them are *provebygeneralization*, *provebyinstantiation*, and *instantiate*. When a state delta with a non-empty mod list is applied, the system attempts to determine which formulas should remain in the usablequantifiers list. This is done mainly by means of a heuristic procedure, the most useful part of which works for formulas of the form

[6]What we propose, however, is not a general solution but only one specific to this program. We are suggesting that a full mathematical description of the final program output be developed, which should not be difficult in this case. But in general, it might not be easy at all: consider for example, a linear programming routine that uses, besides the usual input/output array, an additional workspace to be supplied by the calling program. It is mathematically routine to describe the final contents of the array, but it appears a formidable job to specify precisely the final contents of the workspace.

```
forall v ( < antecedent formula > --> < consequent formula > )
```

Corresponding to the list of usablequantifiers, the system maintains a list of patterns of the antecedent formulas; these are obtained from the actual antecedents by substituting a "*" for the quantified variable, to serve as a place marker. Then if an array element, say `x[.i]`, is covered by the mod list of the state delta being applied, and the result of substituting .i for the * in an antecedent pattern simplifies to "`true`," and there actually are array references to `x` in the corresponding quantified formula, then that formula is deleted from the usablequantifiers list. There are other considerations in this process, but the above is the general idea.

Let us define the *support* of a quantified formula as the smallest set of places upon which the truth of the formula depends—or, to put it another way, such that the truth of the formula is independent of the values of places not in the set. The existing heuristics try to approximate the support of quantified formulas. They do this in a very conservative way, which often results in the deletion of formulas that could be retained. Unfortunately, there is no convenient way for the user to override this, or to give a proof of exactly what the support of a quantified formula is. Improving this situation would be an important enhancement to SDVS.

The present proof actually does indicate a solution to the problem. Precisely because some formulas were being deleted unnecessarily, we worked around the problem by creating and proving state deltas of form

```
[sd pre:  (true)
    comod:  ( support )
    mod:    ()
    post:   ( quantified formula )]
```

Since this static state delta makes the support of the formula explicit, it is retained on the usable list across the application of any other state delta whose mod list is known not to intersect the "support." This is not a complete solution, since to prove the state delta itself, we obviously took advantage of a quirk in the way the system was handling quantifier deletion: after proving the quantified formula, we opened the proof of the state delta above, and found that the quantified formula was still usable. This is logically correct: the formula should be retained since its support is covered by the comod list of the state delta to be proven; but we call it a quirk because the same behavior does not occur when an adatr state delta with a mod list disjoint from the support of the formula is applied. Now suppose all quantified formulas were kept in this way, i.e., as the postconditions of static state deltas such as the one above. Then all the formulas necessary to prove a given state delta could also be made available when it is opened; one would only need to apply the various static state deltas. Some preliminary testing of this scheme has been done, and it would probably work. But it hardly qualifies as convenient. We should try to devise a means of building corresponding capabilities into SDVS without requiring the user to handle the support of a quantifier in such a roundabout way. For example, heuristics could be used to guess the support of a quantifier, and commands could be made available to the user to determine that support and to change it if necessary.

In summary, we have demonstrated SDVS' capability to respond to verification problems for at least some operation programs. The enhancement to handle `for`-loops has been incorporated into the system and demonstated. We have shown how SDVS can be used to approach some types of safety and security problems. Finally, we have evolved some ideas for improvements in SDVS' quantifier-handling capabilities.

References

[1] T. A. Aiken, J. V. Cook, and L. G. Marcus, "Example Proofs of Core Ada Programs in the State Delta Verification System," Technical Report ATR-88(3778)-6, The Aerospace Corporation, 1988.

[2] J. V. Cook and J. E. Doner, "A Modular Correctness Proof of a Quicksort Procedure Written in Ada using SDVS," Technical Report ATR-91(6778)-8, The Aerospace Corporation, 1991.

A. Original Program, Proof, and Proof-trace

As we explained in the introduction, the code as it came to us was a fragment which needed to be embedded in a testing program, in order to define an environment within which the fragment is intelligible. In addition, we simplified the identifier names, and encapsulated the fragment within a procedure, as required for SDVS verification. In the following program, `testctrsort` is the name of the testing program, and the body of procedure `ctrsort` (standing for "contractor sort") is the fragment of interest. Although it is cosmetically different from the code we received, we still refer to this as the "original" program:

```
procedure testctrsort is

        type R_TYPE is
                record
                        a   : integer;
                        b   : integer;
                        c   : integer;
                end record;

        SS: constant integer := 100;

        type ARRAY_TYPE is array (0..SS-1) of R_TYPE;
        X :  ARRAY_TYPE;

        procedure ctrsort(N, S : IN integer) is
                i : integer;
                temp : R_TYPE;
        begin
                -- contractor sort
                for j in 0..N-1 loop
                        i := j;
                        while i < S-1 loop
                                i := i+1;
                                if X(j).b > X(i).b then
                                        temp:=X(j);
                                        X(j):=X(i);
                                        X(i):=temp;
                                end if;
                        end loop;
                end loop;
        end ctrsort;

begin
        null;
end testctrsort;
```

The `ctrsort` procedures sorts an array of records, using the second field of the records as the sorting key. However, the fact that records rather than single integers are involved is irrelevant to the program control structure. Therefore we created a further simplification by replacing the array of records with an array of integers; this was procedure `pblsort` (for "partial bubble sort") and its enclosing program `testpblsort`. The proof discussed in this report was actually a proof for `pblsort`, not the original program. But it is easy to modify the pblsort proof to get one for our original program. It can be obtained from the proof for the pblsort procedure by making systematic changes with a text editor; almost all of these changes simply replace array references of form x[i] by references to record fields of form record(x[i],b). The following is the listing of the resulting correctness proof for the ctrsort procedure:

```
          ; -*- Syntax: Common-lisp; Package: USER; Mode: LISP -*-~%

   (defproof ctrsort.proof
     "(date,
      adatr \"~/ctrsort/testctrsort.a\",
      interpret ctrsort.create.lemma,
      readaxioms \"axioms/arraycoverings.axioms\",
      readaxioms \"axioms/origin-arrays.axioms\",
      date,
      interpret ctrsort.prove.lemma,
      date)")

   (defproof ctrsort.create.lemma
     "(createadalemma ctrsort.lemma
                  file: \"~/ctrsort/testctrsort.a\"
             procedure: ctrsort
        qualified name: testctrsort.ctrsort
          precondition: (range(x) = .s,.s gt 0,.n gt 0,.n le .s,origin(x) = 0)
              mod list: (x)
         postcondition: (forall p (0 le p & p lt .n - 1
                                   --> #record(x[p],b) le #record(x[p + 1],b)),
                         forall q (.n - 1 lt q & q lt .s
                                   --> #record(x[.n - 1],b) le #record(x[q],b))))")

   (defproof ctrsort.prove.lemma
     "(proveadalemma ctrsort.lemma
       proof:
         (applydecls,
          apply u(1),
          apply u(2),
          apply u(1),
          letsd j0.i.loop.exit.sd = u(1),
          letsd j0.i.loop.step.sd = u(2),
          let j0.i.induct.universe = .testctrsort,
          induct on:     .i
            from:        0
            to:          .s - 1
            invariants:  (formula(j0.i.loop.exit.sd),formula(j0.i.loop.step.sd),
                          forall q (0 lt q & q le .i --> .record(x[0],b) le .record(x[q],b)),
                          covering(j0.i.induct.universe,.testctrsort))
            comodlist:   (diff(j0.i.induct.universe,union(testctrsort\pc,i,temp,x[0:(.s - 1)])))
            modlist:     (testctrsort\pc,i,temp,x[0:(.s - 1)])
            base proof:
            step proof:
              (apply u(1),
               apply u(1),
               cases .record(x[0],b) gt .record(x[.i],b)
                 then proof:
                   (apply u(2),
          apply u(1),
          provebygeneralization forall q (0 lt q & q lt .i
                                          --> .record(temp,b) le .record(x[q],b))
            using: (q(1)),
          provebyaxiom alldisjoint(x[1:(.i - 1)],x[0])
            using: disjoint\slice\element,
          prove [sd pre: (true)
                  comod: (temp,i,x[1:(.i - 1)])
                   post: (forall q (0 lt q & q lt .i
                                          --> .record(temp,b) le .record(x[q],b)))]
            proof: ,
          apply u(2),
          apply u(2),
          provebyaxiom alldisjoint(x[1:(.i - 1)],x[.i])
            using: disjoint\slice\element,
          apply u(1),
          apply u(3),
          provebygeneralization g(3)
            using: (q(1))
        else proof:
          (apply u(2),
          provebygeneralization g(3)
            using: (q(1)))),
apply u(2),
apply u(1),
letsd j.loop.exit.sd = u(1),
letsd j.loop.step.sd = u(2),
let j.induct.universe = .testctrsort,
induct on:     .j
  from:        1
  to:          .n
  invariants:  (formula(j.loop.exit.sd),formula(j.loop.step.sd),
                forall p (0 le p & p lt .j - 1
                              --> .record(x[p],b) le .record(x[p + 1],b)),
                forall q (.j le q & q lt .s
                              --> .record(x[.j - 1],b) le .record(x[q],b)),
                covering(j.induct.universe,.testctrsort))
  comodlist:   (diff(j.induct.universe,union(j,i,x,temp,testctrsort\pc)))
  modlist:     (j,i,x,temp,testctrsort\pc)
  base proof:  provebygeneralization g(4)
                 using: (q(1))
  step proof:
    (apply u(1),
     apply u(1),
     letsd i.loop.exit.sd = u(1),
     letsd i.loop.step.sd = u(2),
     let i.induct.universe = .testctrsort,
     induct on:     .i
       from:        .j
       to:          .s - 1
       invariants:  (formula(i.loop.exit.sd),formula(i.loop.step.sd),
                     forall p (0 le p & p lt .j - 1
                                   --> .record(x[p],b) le .record(x[p + 1],b)),
                     forall q (.j le q & q lt .s
                                   --> .record(x[.j - 1],b) le .record(x[q],b)),
               forall q (.j lt q & q le .i
                                 --> .record(x[.j],b) le .record(x[q],b)),
               covering(i.induct.universe,.testctrsort))
comodlist:    (diff(i.induct.universe,
                        union(testctrsort\pc,i,temp,x[.j:(.s - 1)])))
modlist:      (testctrsort\pc,i,temp,x[.j:(.s - 1)])
base proof:
step proof:
  (apply u(1),
   apply u(1),
   cases .record(x[.j],b) gt .record(x[.i],b)
     then proof:
       (apply u(2),
        apply u(1),
        provebyaxiom pcovering(x[0:(.j - 1)],x[.j - 1])
          using: pcovering\slice\element,
        provebyinstantiation .record(x[.j - 1],b) le .record(x[.j],b)
          using: q(2)
          substitutions: (q=.j),
        provebyinstantiation .record(x[.j - 1],b) le .record(x[.i],b)
          using: q(2)
          substitutions: (q=.i),
        provebyaxiom alldisjoint(x[(.i + 1):(.s - 1)],x[.j])
          using: disjoint\slice\element,
        provebyaxiom alldisjoint(x[(.j + 1):(.i - 1)],x[.j])
          using: disjoint\slice\element,
        provebyaxiom alldisjoint(x[0:(.j - 1)],x[.j])
          using: disjoint\slice\element,
        provebygeneralization forall q ((.j lt q & q lt .s) & q ~= .i
                                 --> .record(x[.j - 1],b) le .record(x[q],b))
          using: (q(2)),
        provebygeneralization forall q (.j lt q & q lt .i
                                 --> .record(temp,b) le .record(x[q],b))
          using: (q(2)),
        prove [sd pre: (true)
                comod: (j,s,i,temp,x[0:(.j - 1)],
                        x[(.j + 1):(.i - 1)],
                        x[(.i + 1):(.s - 1)])
                 post: (forall p (0 le p & p lt .j - 1
                                 --> .record(x[p],b) le .record(x[p + 1],b)),
                        forall q ((.j lt q & q lt .s) & q ~= .i
                                 --> .record(x[.j - 1],b) le .record(x[q],b)),
                        forall q (.j lt q & q lt .i
                                 --> .record(temp,b) le .record(x[q],b)))]
          proof: ,
        provebyaxiom pcovering(x[.j:(.s - 1)],x[.j])
          using: pcovering\slice\element,
        apply u(2),
        provebyaxiom pcovering(x[.j:(.s - 1)],x[.i])
          using: pcovering\slice\element,
        provebyaxiom alldisjoint(x[0:(.j - 1)],x[.i])
          using: disjoint\slice\element,
        provebyaxiom alldisjoint(x[(.i + 1):(.s - 1)],x[.i])
          using: disjoint\slice\element,
        provebyaxiom alldisjoint(x[(.j + 1):(.i - 1)],x[.i])
```

```
                    using: disjoint\slice\element,
                  apply u(1),
                  apply u(3),
                  provebygeneralization g(5)
                    using: (q(1)),
                  provebygeneralization g(4)
                    using: (q(3)))
               else proof:
                 (go,
                  provebygeneralization g(5)
                    using: (q(1)))),
        go,
        subcases .j - 1 gt 0
          modlist:
          subgoal:   (forall p (0 le p & p lt .j - 1
                                    --> .record(x[p],b) le .record(x[p + 1],b)))
          then proof:
            (provebyinstantiation .record(x[.j - 2],b) le .record(x[.j - 1],b)
               using: q(2)
               substitutions: (q=.j - 1),
             provebygeneralization g(1)
               using: (q(3)))
          else proof: ,
        provebygeneralization g(4)
          using: (q(2))),
   provebygeneralization g(2)
     using: (q(1)),
   go #testctrsort\pc = exited(testctrsort.ctrsort)))")
```

```
(defaxiom disjoint\slice\element
   "(disjointarray(a) & (m gt i or i gt n) ß alldisjoint(a[m:n],a[i]))"
   (a i m n) nil (slice element) (disjointarray alldisjoint implies))
```

Although the proof for the ctrsort program just presented was obtained from the proof for the pblsort program simply by making systematic changes with a text editor, it nevertheless executes without any trouble. Here follows the trace of the proof obtained with the ctrsort.proof above:

```
=============================================================
Starting image 'sdvs11a'
  with no arguments
  in directory '/u/versys/'
  on machine 'armadillo.aero.org'.

Allegro CL 4.1 [SPARC; R1] (4/2/92 16:26)

;; Copyright Franz Inc., Berkeley, CA, USA
;; Unpublished.  All rights reserved under the copyright laws
;; of the United States.

;; Restricted Rights Legend
;; ------------------------
;; Use, duplication, and disclosure by the Government are subject to
;; restrictions of Restricted Rights for Commercial Software developed
;; at private expense as specified in DOD FAR 52.227-7013 (c) (1) (ii).

;; Starting socket daemon and emacs-lisp interface...
; Loading /u/versys/sdvs/patches11.lisp.

;   Fast loading /u/versys/sdvs/languages/ada/ada3/phase1/temp/adatr1-implementation.fasl.
;   Fast loading /u/versys/sdvs/languages/ada/ada3/phase2/temp/adatr2-implementation.fasl.
;   Fast loading /u/versys/sdvs/languages/ada/ada3/phase2/temp/ada-print.fasl.
;   Fast loading /u/versys/sdvs/languages/vhdl/vhdl-query.fasl.


******************************************************************
* Copyright 1992 The Aerospace Corporation.                      *
* This material may be reproduced by or for the U. S. Government *
* under the clause at DFARS 52.227-7013 (May 1981).              *
******************************************************************

******************************************************************
* Distribution statement: Use or distribution of the State Delta *
* Verification System (SDVS 11) must have prior approval of      *
* DIRNSA, 9800 Savage Road, Ft. George G. Meade, MD 20755-6000   *
* Attention: NSA/R232.                                           *
******************************************************************


Restricted to authorized users only.

<sdvs.1> read
   path name[testproofs/foo.proofs]: /u/doner/ctrsort/ctrsort.proofs

 Definitions read from file "/u/doner/ctrsort/ctrsort.proofs"
   -- (ctrsort.proof,ctrsort.create.lemma,ctrsort.prove.lemma,
       disjoint\slice\element)

<sdvs.2> init
   proof name[]: ctrsort.proof
Warning: 5372656 bytes have been tenured, next gc will be global.
See the documentation for variable *GLOBAL-GC-BEHAVIOR* for more information.

State Delta Verification System, Version 11

Restricted to authorized users only.

 date -- 7/25/92 17:29:47  Elapsed time is 0 seconds.

 Reading parse tree file for Stage 3 Ada file -- "testctrsort.a"

 Translating Stage 3 Ada file -- "/u/doner/ctrsort/testctrsort.a"

 createadalemma -- [sd pre: (.testctrsort\pc = at(testctrsort.ctrsort),
                             range(x) = .s,.s gt 0,.n gt 0,.n le .s,
                             origin(x) = 0)
                       comod: (all)
                         mod: (testctrsort\pc,x)
                        post: (forall p (0 le p & p lt .n - 1
                                  --> #record(x[p],b)
                                         le #record(x[p + 1],b)),
                    forall q (.n - 1 lt q & q lt .s
                                  --> #record(x[.n - 1],b)
                                         le #record(x[q],b)),
                    #testctrsort\pc = exited(testctrsort.ctrsort))]

readaxioms "axioms/arraycoverings.axioms"
  -- (pcovering\slice\element,pcovering\slice\slice,pcovering\element,
      pcovering\slice,disjoint\elements,disjoint\slices,
      disjoint\adjacent\slices)

readaxioms "axioms/origin-arrays.axioms"
  -- (emptyslice,lowerslice,upperslice,totalslice,slicerange,sliceorigin,
      adjacentslices,elementofslice,elementofaconc1,elementofaconc2,
      sliceofaconc)

date -- 7/25/92 17:29:50  Elapsed time is 3 seconds.

open -- [sd pre: (alldisjoint(testctrsort,.testctrsort),
                  covering(.testctrsort,testctrsort\pc,r_type.a,r_type.b,
                           r_type.c,nn,ss,array_type!first,
                           array_type!last,x,k,stdin,stdin\ctr,stdout,
                           stdout\ctr),
                  declare(stdout\ctr,type(integer)),
                  declare(stdout,type(polymorphic)),
                  declare(stdin\ctr,type(integer)),
                  declare(stdin,type(polymorphic)),
                  declare(k,type(integer)),
                  declare(x,
                          type(array,0,(0 + range(x)) - 1,
                               type(record,a(type(integer)),
                                    b(type(integer)),c(type(integer))))),
                  declare(array_type!last,type(integer)),
                  declare(array_type!first,type(integer)),
                  declare(ss,type(integer)),declare(nn,type(integer)),
                  declare(r_type.c,type(integer)),
                  declare(r_type.b,type(integer)),
                  declare(r_type.a,type(integer)),
                  <adatr ctrsort (n, ...);>)
          comod: (all)
            mod: (all)
           post: ([sd pre: (.testctrsort\pc = at(testctrsort.ctrsort),
                            range(x) = .s,.s gt 0,.n gt 0,.n le .s,
                            origin(x) = 0)
                    comod: (all)
                      mod: (diff(all,
                                 diff(union(testctrsort\pc,r_type.a,
                                            r_type.b,r_type.c,nn,ss,
                                            array_type!first,
                                            array_type!last,x,k,stdin,
                                            stdin\ctr,stdout,stdout\ctr,n,
                                            s),
                                      union(testctrsort\pc,x))))
                     post: (forall p (0 le p & p lt .n - 1
                                            --> #record(x[p],b)
                                                  le #record(x[p + 1],b)),
                                    forall q (.n - 1 lt q & q lt .s
                                            --> #record(x[.n - 1],b)
                                                  le #record(x[q],b)),
                                    #testctrsort\pc = exited(testctrsort.ctrsort))])]

apply -- [sd pre: (true)
          comod: (all)
            mod: (testctrsort\pc,testctrsort)
           post: (alldisjoint(testctrsort,.testctrsort,n,s),
                  covering(#testctrsort,.testctrsort,n,s),
                  declare(n,type(integer)),declare(s,type(integer)),
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (testctrsort\pc,n,s)
           post: (#n = .n,#s = .s,
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (testctrsort\pc)
           post: (#testctrsort\pc = at(testctrsort.ctrsort),
                  <adatr null;>)]

go -- breakpoint reached

open -- [sd pre: (.testctrsort\pc = at(testctrsort.ctrsort),
                  range(x) = .s,.s gt 0,.n gt 0,.n le .s,
                  origin(x) = 0)
         comod: (all)
           mod: (diff(all,
                      diff(union(testctrsort\pc,r_type.a,r_type.b,
                                 r_type.c,nn,ss,array_type!first,
                                 array_type!last,x,k,stdin,stdin\ctr,
                                 stdout,stdout\ctr,n,s),
                           union(testctrsort\pc,x))))
          post: (forall p (0 le p & p lt .n - 1
                           --> #record(x[p],b)
                                 le #record(x[p + 1],b)),
                 forall q (.n - 1 lt q & q lt .s
                           --> #record(x[.n - 1],b) le #record(x[q],
                                                                  b)),
                 #testctrsort\pc = exited(testctrsort.ctrsort))]

 apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,testctrsort)
            post: (alldisjoint(testctrsort,.testctrsort,i),
                   covering(#testctrsort,.testctrsort,i),
                   declare(i,type(integer)),
                   <adatr i : integer>)]
```

```
apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,testctrsort)
            post: (alldisjoint(testctrsort,.testctrsort,temp),
                   covering(#testctrsort,.testctrsort,temp),
                   declare(temp,
                           type(record,a(type(integer)),
                                b(type(integer)),c(type(integer)))),
                   <adatr temp : r_type>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,testctrsort)
            post: (alldisjoint(testctrsort,.testctrsort,j),
                   covering(#testctrsort,.testctrsort,j),
                   declare(j,type(integer)),
                   <adatr j : constant integer := 0>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,j)
            post: (#j = 0,
                   <adatr j : constant integer := 0>)]

applydecls -- declaration elaboration complete.

apply -- [sd pre: (.j le .n - 1)
           comod: (all)
             mod: (testctrsort\pc)
            post: (<adatr for ...>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,i)
            post: (#i = .j,
                   <adatr i := j;>)]

letsd -- j0.i.loop.exit.sd = u(1)

letsd -- j0.i.loop.step.sd = u(2)

let -- j0.i.induct.universe = .testctrsort

induction -- .i from 0 to .s - 1

  open -- [sd pre: (true)
            comod: (all)
             post: ([sd pre: (~(.i lt .s - 1))
                      comod: (all)
                        mod: (testctrsort\pc)
                       post: (<adatr while i < s - 1

                                                   i := i + 1;
                                                   ...
                                                 end loop;>)],
                [sd pre: (.i lt .s - 1)
                  comod: (all)
                    mod: (testctrsort\pc)
                   post: (<adatr while i < s - 1

                                   i := i + 1;
                                   ...
                                 end loop;>)],
                forall q (0 lt q & q le .i
                            --> .record(x[0],b) le .record(x[q],b)),
                covering(j0.i.induct.universe,.testctrsort),
                .i = 0)]

close -- 0 steps/applications

open -- [sd pre: (.i ge 0,.i lt .s - 1,
                [sd pre: (~(.i lt .s - 1))
                  comod: (all)
                    mod: (testctrsort\pc)
                   post: (<adatr while i < s - 1

                                   i := i + 1;
                                   ...
                                 end loop;>)],
                [sd pre: (.i lt .s - 1)
                  comod: (all)
                    mod: (testctrsort\pc)
                   post: (<adatr while i < s - 1

                                   i := i + 1;
                                   ...
                                 end loop;>)],
                forall q (0 lt q & q le .i
                            --> .record(x[0],b) le .record(x[q],b)),
                covering(j0.i.induct.universe,.testctrsort))
          comod: (diff(j0.i.induct.universe,
                       union(testctrsort\pc,i,temp,
                             x[0:(.s - 1)])))
            mod: (testctrsort\pc,i,temp,x[0:(.s - 1)])
           post: ([sd pre: (~(.i lt .s - 1))
                    comod: (all)
                      mod: (testctrsort\pc)
                     post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                   end loop;>)],
                  [sd pre: (.i lt .s - 1)
                    comod: (all)
                      mod: (testctrsort\pc)
                     post: (<adatr while i < s - 1

                                     i := i + 1;
                                     ...
                                   end loop;>)],
                forall q (0 lt q & q le #i
                            --> #record(x[0],b) le #record(x[q],b)),
                covering(j0.i.induct.universe,#testctrsort),
                #i = .i + 1)]

apply -- [sd pre: (.i lt .s - 1)
           comod: (all)
             mod: (testctrsort\pc)
            post: (<adatr while i < s - 1

                                i := i + 1;
                                ...
                              end loop;>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,i)
            post: (#i = .i + 1,
                   <adatr i := i + 1;>)]

cases -- .record(x[0],b) gt .record(x[.i],b)

  open -- [sd pre: (.record(x[0],b) gt .record(x[.i],b))
            comod: (all)
              mod: (testctrsort\pc,i,temp,x[0:(s\13 - 1)])
             post: ([sd pre: (~(.i lt .s - 1))
                      comod: (all)
                        mod: (testctrsort\pc)
                       post: (<adatr while i < s - 1

                                          i := i + 1;
                                          ...
                                        end loop;>)],
                    [sd pre: (.i lt .s - 1)
                      comod: (all)
                        mod: (testctrsort\pc)
                       post: (<adatr while i < s - 1

                                          i := i + 1;
                                          ...
                                        end loop;>)],
                    forall q (0 lt q & q le #i
                                --> #record(x[0],b)
                                        le #record(x[q],b)),
                    covering(j0.i.induct.universe,#testctrsort),
                    #i = i\66 + 1)]

    apply -- [sd pre: (.record(x[.j],b) gt .record(x[.i],b),
                        .j ge origin(x),
                        .j le (origin(x) + range(x)) - 1,
                        .i ge origin(x),
                        .i le (origin(x) + range(x)) - 1)
               comod: (all)
                 mod: (testctrsort\pc)
                post: (<adatr if x (j).b > x (i).b
                                then temp := x (j);
                                              ...
                             end if;>)]

apply -- [sd pre: (.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testctrsort\pc,temp)
            post: (#temp = .x[.j],
                   <adatr temp := x (j);>)]

provebygeneralization -- forall q (0 lt q &
                                   q lt 1 + i\66
                                   --> foo\83
                                         le .record(x[q],b))

provebyaxiom disjoint\slice\element -- alldisjoint(x[1
                                                     :(.i - 1)],
                                                   x[0])

open -- [sd pre: (true)
          comod: (temp,i,x[1:(.i - 1)])
           post: (forall q (0 lt q & q lt .i
                            --> .record(temp,b)
                                  le .record(x[q],b)))]

close -- 0 steps/applications

apply -- [sd pre: (.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1,
                   .i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testctrsort\pc,x[.j])
            post: (#x[.j] = .x[.i],
                   <adatr x (j) := x (i);>)]

apply -- [sd pre: (true)
           comod: (temp,i,x[1:((1 + i\66) - 1)])
            post: (forall q (0 lt q & q lt .i
                             --> .record(temp,b)
                                   le .record(x[q],b)))]

provebyaxiom disjoint\slice\element -- alldisjoint(x[1
                                                     :(.i - 1)],
                                                   x[.i])

apply -- [sd pre: (.i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testctrsort\pc,x[.i])
            post: (#x[.i] = .temp,
                   <adatr x (i) := temp;>)]

apply -- [sd pre: (true)
```

```
            comod: (temp,i,x[1:((1 + i\66) - 1)])
             post: (forall q (0 lt q & q lt .i
                                  --> .record(temp,b)
                                        le .record(x[q],b)))]

  provebygeneralization -- forall q (0 lt q &
                                         q le 1 + i\66
                                         --> foo\92
                                               le .record(x[q],b))

close -- 11 steps/applications

open -- [sd pre: (~(.record(x[0],b) gt .record(x[.i],b)))
           comod: (all)
             mod: (testctrsort\pc,i,temp,x[0:(s\13 - 1)])
            post: ([sd pre: (~(.i lt .s - 1))
                      comod: (all)
                        mod: (testctrsort\pc)
                       post: (<adatr while i < s - 1

                                           i := i + 1;
                                           ...
                                         end loop;>)],
                    [sd pre: (.i lt .s - 1)
                      comod: (all)
                        mod: (testctrsort\pc)
                       post: (<adatr while i < s - 1

                                           i := i + 1;
                                           ...
                                         end loop;>)],
                    forall q (0 lt q & q le #i
                                --> #record(x[0],b)
                                      le #record(x[q],b)),
                    covering(j0.i.induct.universe,#testctrsort),
                    #i = i\66 + 1)]

  apply -- [sd pre: (~(.record(x[.j],b) gt .record(x[.i],b)),
                     .j ge origin(x),
                     .j le (origin(x) + range(x)) - 1,
                     .i ge origin(x),
                     .i le (origin(x) + range(x)) - 1)
            comod: (all)
              mod: (testctrsort\pc)
             post: (<adatr if x (j).b > x (i).b
                                  then temp := x (j);
                                         ...
                               end if;>)]

  provebygeneralization -- forall q (0 lt q &
                                         q le 1 + i\66
                                         --> foo\67
                                               le .record(x[q],b))

close -- 2 steps/applications
```

```
      join -- [sd pre: (true)
               comod: (all)
                 mod: (testctrsort\pc,i,temp,x[0:(s\13 - 1)])
                post: ([sd pre: (~(.i lt .s - 1))
                          comod: (all)
                            mod: (testctrsort\pc)
                           post: (<adatr while i < s - 1

                                                i := i + 1;
                                                ...
                                              end loop;>)],
                        [sd pre: (.i lt .s - 1)
                          comod: (all)
                            mod: (testctrsort\pc)
                           post: (<adatr while i < s - 1

                                                i := i + 1;
                                                ...
                                              end loop;>)],
                        forall q (0 lt q & q le #i
                                    --> #record(x[0],b) le #record(x[q],
                                                                     b)),
                        covering(j0.i.induct.universe,#testctrsort),
                        #i = i\66 + 1)]

      inserting -- pcovering(all,x[0:(s\13 - 1)])

      inserting -- pcovering(all,x[0:(s\13 - 1)])

    close -- 3 steps/applications

  join induction cases -- [sd pre: (0 le .s - 1)
                             comod: (all,
                                     diff(j0.i.induct.universe,
                                          union(testctrsort\pc,i,temp,
                                                x[0:(s\13 - 1)])))
                               mod: (testctrsort\pc,i,temp,
                                     x[0:(.s - 1)])
                              post: (#i = .s - 1,
                                     formula(j0.i.loop.exit.sd),
                                     formula(j0.i.loop.step.sd),
                                     forall q (0 lt q & q le #i
                                                --> #record(x[0],b)
                                                     le #record(x[q],
                                                                  b)),
                                     covering(j0.i.induct.universe,
                                              #testctrsort))]

  inserting -- pcovering(all,x[0:(s\13 - 1)])

  inserting -- pcovering(all,x[0:(s\13 - 1)])

  apply -- [sd pre: (~(.i lt .s - 1))
              comod: (all)
```

```
            mod: (testctrsort\pc)
           post: (<adatr while i < s - 1

                                   i := i + 1;
                                   ...
                                 end loop;>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,j)
            post: (#j = .j + 1,
                   <adatr j := j + 1;>)]

letsd -- j.loop.exit.sd = u(1)

letsd -- j.loop.step.sd = u(2)

let -- j.induct.universe = .testctrsort

induction -- .j from 1 to .n

  open -- [sd pre: (true)
            comod: (all)
             post: ([sd pre: (.j gt .n - 1)
                      comod: (all)
                        mod: (testctrsort\pc)
                       post: (<adatr for ...>)],
                    [sd pre: (.j le .n - 1)
                      comod: (all)
                        mod: (testctrsort\pc)
                       post: (<adatr for ...>)],
                    forall p (0 le p & p lt .j - 1
                              --> .record(x[p],b)
                                    le .record(x[p + 1],b)),
                    forall q (.j le q & q lt .s
                              --> .record(x[.j - 1],b)
                                    le .record(x[q],b)),
                    covering(j.induct.universe,.testctrsort),
                    .j = 1)]

    provebygeneralization -- forall q (1 le q & q lt s\13
                                       --> foo\129 le .record(x[q],
                                                              b))

  close -- 1 steps/applications

  open -- [sd pre: (.j ge 1,.j lt .n,
                    [sd pre: (.j gt .n - 1)
                      comod: (all)
                        mod: (testctrsort\pc)
                       post: (<adatr for ...>)],
                    [sd pre: (.j le .n - 1)
                      comod: (all)
                        mod: (testctrsort\pc)
                       post: (<adatr for ...>)],
                forall p (0 le p & p lt .j - 1
                            --> .record(x[p],b)
                                  le .record(x[p + 1],b)),
                forall q (.j le q & q lt .s
                            --> .record(x[.j - 1],b)
                                  le .record(x[q],b)),
                covering(j.induct.universe,.testctrsort))
        comod: (diff(j.induct.universe,
                     union(j,i,x,temp,testctrsort\pc)))
          mod: (j,i,x,temp,testctrsort\pc)
         post: ([sd pre: (.j gt .n - 1)
                  comod: (all)
                    mod: (testctrsort\pc)
                   post: (<adatr for ...>)],
                [sd pre: (.j le .n - 1)
                  comod: (all)
                    mod: (testctrsort\pc)
                   post: (<adatr for ...>)],
                forall p (0 le p & p lt #j - 1
                            --> #record(x[p],b)
                                  le #record(x[p + 1],b)),
                forall q (#j le q & q lt #s
                            --> #record(x[#j - 1],b)
                                  le #record(x[q],b)),
                covering(j.induct.universe,#testctrsort),
                #j = .j + 1)]

inserting -- pcovering(all,record(x[j\184 - 1],b))

inserting -- pcovering(x,x[j\184 - 1])

apply -- [sd pre: (.j le .n - 1)
           comod: (all)
             mod: (testctrsort\pc)
            post: (<adatr for ...>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,i)
            post: (#i = .j,
                   <adatr i := j;>)]

letsd -- i.loop.exit.sd = u(1)

letsd -- i.loop.step.sd = u(2)

let -- i.induct.universe = .testctrsort

induction -- .i from .j to .s - 1

  open -- [sd pre: (true)
            comod: (all)
             post: ([sd pre: (~(.i lt .s - 1))
                      comod: (all)
                        mod: (testctrsort\pc)
                   post: (<adatr while i < s - 1

                                    i := i + 1;
                                    ...
                                  end loop;>)],
                [sd pre: (.i lt .s - 1)
                  comod: (all)
                    mod: (testctrsort\pc)
                   post: (<adatr while i < s - 1

                                    i := i + 1;
                                    ...
                                  end loop;>)],
                forall p (0 le p & p lt .j - 1
                             --> .record(x[p],b)
                                   le .record(x[p + 1],b)),
                forall q (.j le q & q lt .s
                             --> .record(x[.j - 1],b)
                                   le .record(x[q],b)),
                forall q (.j lt q & q le .i
                             --> .record(x[.j],b)
                                   le .record(x[q],b)),
                covering(i.induct.universe,.testctrsort),
                .i = .j)]

close -- 0 steps/applications

open -- [sd pre: (.i ge .j,.i lt .s - 1,
                [sd pre: (~(.i lt .s - 1))
                  comod: (all)
                    mod: (testctrsort\pc)
                   post: (<adatr while i < s - 1

                                    i := i + 1;
                                    ...
                                  end loop;>)],
                [sd pre: (.i lt .s - 1)
                  comod: (all)
                    mod: (testctrsort\pc)
                   post: (<adatr while i < s - 1

                                    i := i + 1;
                                    ...
                                  end loop;>)],
                forall p (0 le p & p lt .j - 1
                             --> .record(x[p],b)
                                   le .record(x[p + 1],b)),
                forall q (.j le q & q lt .s
                             --> .record(x[.j - 1],b)
                                   le .record(x[q],b)),
                forall q (.j lt q & q le .i
                             --> .record(x[.j],b)
                                   le .record(x[q],b)),
                covering(i.induct.universe,.testctrsort))
          comod: (diff(i.induct.universe,
                              union(testctrsort\pc,i,temp,
                                    x[.j:(.s - 1)])))
                 mod: (testctrsort\pc,i,temp,x[.j:(.s - 1)])
                post: ([sd pre: (~(.i lt .s - 1))
                          comod: (all)
                            mod: (testctrsort\pc)
                           post: (<adatr while i < s - 1

                                         i := i + 1;
                                         ...
                                       end loop;>)],
                        [sd pre: (.i lt .s - 1)
                          comod: (all)
                            mod: (testctrsort\pc)
                           post: (<adatr while i < s - 1

                                         i := i + 1;
                                         ...
                                       end loop;>)],
                        forall p (0 le p & p lt #j - 1
                                    --> #record(x[p],b)
                                            le #record(x[p + 1],b)),
                        forall q (#j le q & q lt #s
                                    --> #record(x[#j - 1],b)
                                            le #record(x[q],b)),
                        forall q (#j lt q & q le #i
                                    --> #record(x[#j],b)
                                            le #record(x[q],b)),
                        covering(i.induct.universe,#testctrsort),
                        #i = .i + 1)]

inserting -- pcovering(all,record(x[j\184 - 1],b))

inserting -- pcovering(x,x[j\184 - 1])

apply -- [sd pre: (.i lt .s - 1)
            comod: (all)
              mod: (testctrsort\pc)
             post: (<adatr while i < s - 1

                           i := i + 1;
                           ...
                         end loop;>)]

apply -- [sd pre: (true)
            comod: (all)
              mod: (testctrsort\pc,i)
             post: (#i = .i + 1,
                    <adatr i := i + 1;>)]

cases -- .record(x[.j],b) gt .record(x[.i],b)

  open -- [sd pre: (.record(x[.j],b) gt .record(x[.i],b))
             comod: (all)
               mod: (testctrsort\pc,i,temp,
              x[j\184:(s\13 - 1)])
      post: ([sd pre: (~(.i lt .s - 1))
                comod: (all)
                  mod: (testctrsort\pc)
                 post: (<adatr while i < s - 1

                                    i := i + 1;
                                    ...
                                  end loop;>)],
              [sd pre: (.i lt .s - 1)
                comod: (all)
                  mod: (testctrsort\pc)
                 post: (<adatr while i < s - 1

                                    i := i + 1;
                                    ...
                                  end loop;>)],
              forall p (0 le p & p lt #j - 1
                          --> #record(x[p],b)
                                 le #record(x[p + 1],
                                                 b)),
              forall q (#j le q & q lt #s
                          --> #record(x[#j - 1],b)
                                 le #record(x[q],b)),
              forall q (#j lt q & q le #i
                          --> #record(x[#j],b)
                                 le #record(x[q],b)),
              covering(i.induct.universe,#testctrsort),
              #i = i\206 + 1)]

apply -- [sd pre: (.record(x[.j],b) gt .record(x[.i],b),
                   .j ge origin(x),
                   .j le (origin(x) + range(x)) - 1,
                   .i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testctrsort\pc)
            post: (<adatr if x (j).b > x (i).b
                             then temp := x (j);
                                  ...
                          end if;>)]

apply -- [sd pre: (.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testctrsort\pc,temp)
            post: (#temp = .x[.j],
                   <adatr temp := x (j);>)]

provebyaxiom pcovering\slice\element -- pcovering(x[0
                                                   :(.j - 1)],
                                                  x[.j - 1])

provebyinstantiation -- foo\212 le foo\217
```

```
                    provebyinstantiation -- foo\212 le foo\225

                    provebyaxiom disjoint\slice\element -- alldisjoint(x[(.i +
                                                                           1)
                                                                        :(.s
                                                                           - 1)],
                                                                      x[.j])

                    provebyaxiom disjoint\slice\element -- alldisjoint(x[(.j +
                                                                           1)
                                                                        :(.i
                                                                           - 1)],
                                                                      x[.j])

                    provebyaxiom disjoint\slice\element -- alldisjoint(x[0
                                                                        :(.j
                                                                           - 1)],
                                                                      x[.j])


  WARNING: the sparse matrix for Coverings has just grown to 1000 columns.



  WARNING: the sparse matrix for Coverings has just grown to 2000 columns.


                    provebygeneralization -- forall q ((j\184 lt q &
                                                          q lt s\13) &
                                                        q ~= 1 + i\206
                                                        --> foo\212
                                                                le .record(x[q],
                                                                           b))
 Warning: 5086800 bytes have been tenured, next gc will be global.
 See the documentation for variable *GLOBAL-GC-BEHAVIOR* for more information.

                    provebygeneralization -- forall q (j\184 lt q &
                                                        q lt 1 + i\206
                                                        --> foo\236
                                                                le .record(x[q],
                                                                           b))

                    open -- [sd pre: (true)
                              comod: (j,s,i,temp,x[0:(.j - 1)],
                                      x[(.j + 1):(.i - 1)],
                                      x[(.i + 1):(.s - 1)])
                               post: (forall p (0 le p & p lt .j - 1
                                                --> .record(x[p],b)
                                                    le .record(x[p + 1],
                                                               b)),
                                      forall q ((.j lt q & q lt .s) &
                                                q ~= .i
                                                --> .record(x[.j - 1],b)
                                                    le .record(x[q],b)),
                                      forall q (.j lt q & q lt .i
                                  --> .record(temp,b)
                                         le .record(x[q],b)))]

close -- 0 steps/applications

provebyaxiom pcovering\slice\element -- pcovering(x[.j
                                                    :(.s - 1)],
                                                  x[.j])

apply -- [sd pre: (.j ge origin(x),
                   .j le (origin(x) + range(x)) - 1,
                   .i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testctrsort\pc,x[.j])
            post: (#x[.j] = .x[.i],
                   <adatr x (j) := x (i);>)]

provebyaxiom pcovering\slice\element -- pcovering(x[.j
                                                    :(.s - 1)],
                                                  x[.i])

provebyaxiom disjoint\slice\element -- alldisjoint(x[0
                                                     :(.j
                                                        - 1)],
                                                   x[.i])

provebyaxiom disjoint\slice\element -- alldisjoint(x[(.i +
                                                       1)
                                                     :(.s
                                                        - 1)],
                                                   x[.i])

provebyaxiom disjoint\slice\element -- alldisjoint(x[(.j +
                                                       1)
                                                     :(.i
                                                        - 1)],
                                                   x[.i])

apply -- [sd pre: (.i ge origin(x),
                   .i le (origin(x) + range(x)) - 1)
           comod: (all)
             mod: (testctrsort\pc,x[.i])
            post: (#x[.i] = .temp,
                   <adatr x (i) := temp;>)]

apply -- [sd pre: (true)
           comod: (j,s,i,temp,x[0:(j\184 - 1)],
                   x[(j\184 + 1)
                     :((1 + i\206) - 1)],
                   x[((1 + i\206) + 1)
                     :(s\13 - 1)])
            post: (forall p (0 le p &
                                p lt .j - 1
                                --> .record(x[p],b)
```

```
                                         le .record(x[p + 1],
                                                    b)),
                     forall q ((.j lt q & q lt .s) &
                                  q ~= .i
                                  --> .record(x[.j - 1],
                                              b)
                                        le .record(x[q],b)),
                     forall q (.j lt q & q lt .i
                                  --> .record(temp,b)
                                        le .record(x[q],b)))]

  provebygeneralization -- forall q (j\184 lt q &
                                       q le 1 + i\206
                                       --> foo\245
                                              le .record(x[q],
                                                         b))

  provebygeneralization -- forall q (j\184 le q &
                                       q lt s\13
                                       --> foo\212
                                              le .record(x[q],
                                                         b))

close -- 21 steps/applications

open -- [sd pre: (~(.record(x[.j],b) gt .record(x[.i],b)))
          comod: (all)
            mod: (testctrsort\pc,i,temp,
                  x[j\184:(s\13 - 1)])
           post: ([sd pre: (~(.i lt .s - 1))
                    comod: (all)
                      mod: (testctrsort\pc)
                     post: (<adatr while i < s - 1

                                         i := i + 1;
                                         ...
                                       end loop;>)],
                  [sd pre: (.i lt .s - 1)
                    comod: (all)
                      mod: (testctrsort\pc)
                     post: (<adatr while i < s - 1

                                         i := i + 1;
                                         ...
                                       end loop;>)],
                  forall p (0 le p & p lt #j - 1
                              --> #record(x[p],b)
                                    le #record(x[p + 1],
                                               b)),
                  forall q (#j le q & q lt #s
                              --> #record(x[#j - 1],b)
                                    le #record(x[q],b)),
                  forall q (#j lt q & q le #i
                              --> #record(x[#j],b)
                                    le #record(x[q],b)),
                    covering(i.induct.universe,#testctrsort),
                    #i = i\206 + 1)]

     apply -- [sd pre: (~(.record(x[.j],b) gt .record(x[.i],b)),
                         .j ge origin(x),
                         .j le (origin(x) + range(x)) - 1,
                         .i ge origin(x),
                         .i le (origin(x) + range(x)) - 1)
                comod: (all)
                  mod: (testctrsort\pc)
                 post: (<adatr if x (j).b > x (i).b
                                 then temp := x (j);
                                        ...
                               end if;>)]

     go -- no more declarations or statements

     provebygeneralization -- forall q (j\184 lt q &
                                        q le 1 + i\206
                                        --> foo\217
                                               le .record(x[q],
                                                          b))

    close -- 2 steps/applications

  join -- [sd pre: (true)
            comod: (all)
              mod: (testctrsort\pc,i,temp,
                    x[j\184:(s\13 - 1)])
             post: ([sd pre: (~(.i lt .s - 1))
                       comod: (all)
                         mod: (testctrsort\pc)
                        post: (<adatr while i < s - 1

                                        i := i + 1;
                                        ...
                                      end loop;>)],
                    [sd pre: (.i lt .s - 1)
                       comod: (all)
                         mod: (testctrsort\pc)
                        post: (<adatr while i < s - 1

                                        i := i + 1;
                                        ...
                                      end loop;>)],
                    forall p (0 le p & p lt #j - 1
                                --> #record(x[p],b)
                                      le #record(x[p + 1],b)),
                    forall q (#j le q & q lt #s
                                --> #record(x[#j - 1],b)
                                      le #record(x[q],b)),
                    forall q (#j lt q & q le #i
                                --> #record(x[#j],b)
                                      le #record(x[q],b)),
                    covering(i.induct.universe,#testctrsort),
                  #i = i\206 + 1)]

     inserting -- pcovering(all,x[j\184:(s\13 - 1)])

     inserting -- pcovering(all,record(x[j\184 - 1],b))

     inserting -- pcovering(x,x[j\184 - 1])

     inserting -- pcovering(all,x[j\184:(s\13 - 1)])

   close -- 3 steps/applications

join induction cases -- [sd pre: (.j le .s - 1)
                           comod: (all,
                                   diff(i.induct.universe,
                                        union(testctrsort\pc,i,
                                              temp,
                                              x[j\184
                                                :(s\13 - 1)])))
                             mod: (testctrsort\pc,i,temp,
                                   x[.j:(.s - 1)])
                            post: (#i = .s - 1,
                                   formula(i.loop.exit.sd),
                                   formula(i.loop.step.sd),
                                   forall p (0 le p &
                                             p lt #j - 1
                                             --> #record(x[p],b)
                                                   le #record(x[p +
                                                                 1],
                                                              b)),
                                   forall q (#j le q & q lt #s
                                             --> #record(x[#j - 1],
                                                           b)
                                                   le #record(x[q],
                                                              b)),
                                   forall q (#j lt q & q le #i
                                             --> #record(x[#j],b)
                                                   le #record(x[q],
                                                              b)),
                                   covering(i.induct.universe,
                                            #testctrsort))]

inserting -- pcovering(all,x[j\184:(s\13 - 1)])

inserting -- pcovering(all,record(x[j\184 - 1],b))

inserting -- pcovering(x,x[j\184 - 1])

inserting -- pcovering(all,x[j\184:(s\13 - 1)])

apply -- [sd pre: (~(.i lt .s - 1))
           comod: (all)
             mod: (testctrsort\pc)
            post: (<adatr while i < s - 1
                                    i := i + 1;
                                    ...
                                  end loop;>)]

      apply -- [sd pre: (true)
                 comod: (all)
                   mod: (testctrsort\pc,j)
                  post: (#j = .j + 1,
                         <adatr j := j + 1;>)]

      go -- no more declarations or statements

      subcases -- .j - 1 gt 0

        open -- [sd pre: (.j - 1 gt 0)
                  comod: (all)
                   post: (forall p (0 le p & p lt .j - 1
                                     --> .record(x[p],b)
                                           le .record(x[p + 1],b)))]

          provebyinstantiation -- foo\304 le foo\309

          provebygeneralization -- forall p (0 le p &
                                             p  lt (1 + j\184) - 1
                                             --> .record(x[p],b)
                                                    le .record(x[p +
                                                                   1],
                                                               b))

        close -- 2 steps/applications

        open -- [sd pre: (~(.j - 1 gt 0))
                  comod: (all)
                   post: (forall p (0 le p & p lt .j - 1
                                     --> .record(x[p],b)
                                           le .record(x[p + 1],b)))]

          The state delta is vacuously TRUE because its precondition is
          FALSE.

        close -- 0 steps/applications

      join -- [sd pre: (true)
                comod: (all)
                 post: (forall p (0 le p & p lt .j - 1
                                   --> .record(x[p],b)
                                         le .record(x[p + 1],b)))]

      provebygeneralization -- forall q (1 + j\184 le q & q lt s\13
                                         --> foo\309 le .record(x[q],
                                                                  b))

    close -- 10 steps/applications

  join induction cases -- [sd pre: (1 le .n)
```

```
                          comod: (all,
                                  diff(j.induct.universe,
                                       union(j,i,x,temp,
                                             testctrsort\pc)))
                            mod: (j,i,x,temp,testctrsort\pc)
                           post: (#j = .n,formula(j.loop.exit.sd),
                                  formula(j.loop.step.sd),
                                  forall p (0 le p &
                                            p lt #j - 1
                                            --> #record(x[p],b)
                                                    le #record(x[p +
                                                                  1],
                                                               b)),
                                  forall q (#j le q & q lt #s
                                            --> #record(x[#j - 1],
                                                        b)
                                                    le #record(x[q],
                                                               b)),
                                  covering(j.induct.universe,
                                           #testctrsort))]

inserting -- pcovering(all,record(x[n\12 - 1],b))

inserting -- pcovering(x,x[n\12 - 1])

provebygeneralization -- forall q (n\12 - 1 lt q & q lt s\13
                                   --> foo\359 le .record(x[q],b))

apply -- [sd pre: (.j gt .n - 1)
           comod: (all)
             mod: (testctrsort\pc)
            post: (<adatr for ...>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,testctrsort,j)
            post: (covering(.testctrsort,#testctrsort,j),
                   undeclare(j),
                   <adatr j : constant integer := 0>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,testctrsort,temp,record(temp,a),
                   record(temp,b),record(temp,c))
            post: (covering(.testctrsort,#testctrsort,temp,
                            record(temp,a),record(temp,b),
                            record(temp,c)),
                   undeclare(temp,record(temp,a),record(temp,b),
                             record(temp,c)),
                   <adatr temp : r_type>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (testctrsort\pc,testctrsort,i)
            post: (covering(.testctrsort,#testctrsort,i),
                          undeclare(i),
                          <adatr i : integer>)]

        apply -- [sd pre: (true)
                   comod: (all)
                     mod: (testctrsort\pc)
                    post: (#testctrsort\pc = exited(testctrsort.ctrsort),
                          <adatr null;>)]

      close -- 22 steps/applications

    close -- 4 steps/applications

    proveadalemma -- [sd pre: (.testctrsort\pc = at(testctrsort.ctrsort),
                                range(x) = .s,.s gt 0,.n gt 0,.n le .s,
                                origin(x) = 0)
                         comod: (all)
                           mod: (testctrsort\pc,x)
                          post: (forall p (0 le p & p lt .n - 1
                                            --> #record(x[p],b)
                                                  le #record(x[p + 1],b)),
                                forall q (.n - 1 lt q & q lt .s
                                            --> #record(x[.n - 1],b)
                                                  le #record(x[q],b)),
                               #testctrsort\pc = exited(testctrsort.ctrsort))]

    date -- 7/26/92 03:07:41  Elapsed time is -9 seconds.
```